Implementation of an AMIDAR-based Java Processor

Implementierung eines AMIDAR-basierten Java Prozessors
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Dipl.-Inform. Changgong Li aus Heilongjiang, China
Tag der Einreichung: 02.10.2018, Tag der Prüfung: 08.04.2019
Darmstadt — D 17

1. Gutachter: Prof. Dr.-Ing. Christian Hochberger
2. Gutachter: Prof. Dr. Wolfgang Karl

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Rechnersysteme
Fachbereich Elektrotechnik und Informationstechnik

Implementation of an AMIDAR-based Java Processor
Implementierung eines AMIDAR-basierten Java Prozessors

Genehmigte Dissertation von Dipl.-Inform. Changgong Li aus Heilongjiang, China

1. Gutachter: Prof. Dr.-Ing. Christian Hochberger
2. Gutachter: Prof. Dr. Wolfgang Karl

Tag der Einreichung: 02.10.2018
Tag der Prüfung: 08.04.2019

Darmstadt — D 17

Li, Changgong: Implementation of an AMIDAR-based Java Processor
Darmstadt, Technische Universität Darmstadt,
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
Tag der mündlichen Prüfung: 08.04.2019

Erklärung laut Promotionsordung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, den 02.10.2018

_____

(Dipl.-Inform. Changgong Li)

Kurzfassung

In dieser Arbeit wird ein Java Prozessor vorgestellt, welcher auf der *Adaptive Microinstruction Driven Architecture* (AMIDAR) basiert. Dieser Prozessor wird bereits als Forschungsplattform zur Untersuchung und Entwicklung adaptiver Prozessor-Architekturen verwendet. Mittels eines konfigurierbaren Beschleunigers ist er in der Lage, zur Laufzeit einer Applikation auf deren spezifische Anforderungen zu reagieren und somit deren Ausführungs-Performance dynamisch zu erhöhen.

Gegenüber klassischen RISC-Prozessoren besteht ein AMIDAR-basierter Prozessor aus vier unterschiedlichen Arten von Komponenten: einer Token-Machine, verschiedenen funktionalen Einheiten (FUs), einem Token-Verteilungsnetzwerk sowie einer FU-Kommunikationsstruktur. Die Token-Machine ist eine spezielle FU, welche die Ausführungen der anderen FUs steuert. Dazu übersetzt sie die Instruktionen in einen Satz von Mikroinstruktionen, den sogenannten Tokens, und sendet diese über das Token-Verteilungsnetzwerk an die entsprechenden FUs. Die Tokens teilen einzelnen FUs mit, welche Operationen auf den Eingangsdaten ausgeführt werden und an welche FUs die Ergebnisse anschließend geschickt werden sollen. Nachdem eine Operation ausgeführt wurde, wird deren Ergebnis an die FU-Kommunikationsstruktur übergeben, damit es an die vorgegebene Ziel-FU weitergeleitet werden kann.

Für den Instruktionssatz, welcher durch den Java-Bytecode definiert ist, sind insgesamt sechs FUs mit bestimmten Funktionalitäten für den Java Prozessor entwickelt worden. Diese umfassen einen Frame Stack, einen Heap Manager, einen Thread Scheduler, einen Debugger, eine Integer-ALU und eine Floating-Point Unit. Mit diesen FUs kann der Prozessor bereits die SPEC JVM98 Benchmarks fehlerfrei durchführen. Dies deutet darauf hin, dass er sich über eingebettete Software hinaus für ein breites Spektrum von Anwendungen einsetzen lässt.

Neben der Bytecode-Ausführung beinhaltet dieser Prozessor auch einige erweiterte Funktionen, welche seine Leistung und Nutzbarkeit deutlich verbessert haben. Zum Ersten enthält er einen Objekt-Cache basierend auf einer neuartigen Methode zur Generierung der Cache-Indizes, welche eine bessere durchschnittliche Trefferrate bietet, als die klassische XOR-basierte Methode. Zum Zweiten ist ein hardwarebasierter Garbage Collector in den Heap Manager integriert, welcher den durch den Garbage Collection Prozess verursachten Overhead erheblich reduzieren kann. Zum Dritten ist die Thread-Verwaltung ebenfalls komplett in Hardware umgesetzt und kann deshalb parallel mit der laufenden Anwendung durchgeführt werden. Außerdem ist ein Debugging Framework für den Prozessor entwickelt worden, welches mehrere mächtige Debugging-Funktionalitäten auf Hardware- und Software-Ebene bereitstellt.

Abstract

This thesis presents a Java processor based on the *Adaptive Microinstruction Driven Architecture* (AMIDAR). This processor is intended as a research platform for investigating adaptive processor architectures. Combined with a configurable accelerator, it is able to detect and speed up hot spots of arbitrary applications dynamically.

In contrast to classical RISC processors, an AMIDAR-based processor consists of four main types of components: a token machine, functional units (FUs), a token distribution network and an FU interconnect structure. The token machine is a specialized functional unit and controls the other FUs by means of tokens. These tokens are delivered to the FUs over the token distribution network. The tokens inform the FUs about what to do with input data and where to send the results. Data is exchanged among the FUs over the FU interconnect structure.

Based on the virtual machine architecture defined by the Java bytecode, a total of six FUs have been developed for the Java processor, namely a frame stack, a heap manager, a thread scheduler, a debugger, an integer ALU and a floating-point unit. Using these FUs, the processor can already execute the SPEC JVM98 benchmark suite properly. This indicates that it can be employed to run a broad variety of applications rather than embedded software only.

Besides bytecode execution, several enhanced features have also been implemented in the processor to improve its performance and usability. First, the processor includes an object cache using a novel cache index generation scheme that provides a better average hit rate than the classical XOR-based scheme. Second, a hardware garbage collector has been integrated into the heap manager, which greatly reduces the overhead caused by the garbage collection process. Third, thread scheduling has been realized in hardware as well, which allows it to be performed concurrently with the running application. Furthermore, a complete debugging framework has been developed for the processor, which provides powerful debugging functionalities at both software and hardware levels.

Contents

List of Figures

List of Tables

List of Abbreviations

ADLA	Abstract Description Language for AMIDAR Processors
ADP	AMIDAR Debug Protocol
AMAT	Average Memory Access Time
AMIDAR	Adaptive Microinstruction Driven Architecture
AMP	Active Memory Processor
AMTI	Absolute Method Table Index
API	Application Programming Interface
AST	Abstract Syntax Tree
AXI	Advanced eXtensible Interface
AXT	AMIDAR Executable
BNF	Backus-Naur Form
BRAM	Block Random-Access Memory
BTC	Binary Tree of Comparators
CAM	Content-Addressable Memory
CBT	Complete Binary Tree
CCP	Caller Context Pointer
CDC	Connected Device Configuration
CGMT	Coarse Grained Multithreading
CGRA	Coarse Grained Reconfigurable Array
CLDC	Connected Limited Device Configuration
CT	Class Table
CTI	Class Table Index
DEX	Dalvik Executable
DMA	Direct Memory Access
DMS	Dynamic Mask Selection
DRAM	Dynamic Random-Access Memory
EDF	Earliest Deadline First
ETS	Exception Table Section
EUI	Exception Unit Interface
FIFO	First In, First Out
FLC	Flat-Lock
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
FU-NI	Functional Unit Native Interface
GC	Garbage Collection
GCMM	Garbage-Collected Memory Module
GCU	Garbage Collector Unit
GP	Guaranteed Percentage
HDL	Hardware Description Language
ICE	In-Circuit Emulation
IDE	Integrated Development Environment
IIS	Implemented Interfaces Section
IMT	Interleaved Multithreading
IP	Intellectual Property
IRQ	Interrupt Request

ISA	Instruction Set Architecture
IST	Interrupt Service Thread
ITS	Interface Table Section
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
Jamuth	Java Multithreaded Processor
JDI	Java Debug Interface
JDT	Java Development Tools
JOP	Java Optimized Processor
JRE	Java Runtime Environment
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LIFO	Last In, First Out
LILT	Logically-Indexed, Logically-Tagged
LSB	Least Significant Bit
LSTF	Least Slack Time First
LUT	Lookup Table
LVP	Local Variables Pointer
MDM	MicroBlaze Debug Module
MIG	Memory Interface Generator
MLFQ	Multilevel Feedback Queue
MLQ	Multilevel Queue
MMU	Memory Management Unit
MSB	Most Significant Bit
OLAT	Ordered List of Array Types
OLI	Ordered List of Interfaces
OSM	Object Space Manager
PQ	Priority Queue
RANSAC	Random Sample Consensus
RM	Rate Monotonic
RMTI	Relative Method Table Index
RR	Round-Robin
RRA	Round-Robin Arbiter
RTM	Real-Time Task Manager
RTOS	Real-Time Operating System
RTU	Real-Time Unit
SA	Systolic Array
SHAP	Secure Hardware Agent Platform
SMT	Simultaneous Multithreading
SoC	System-on-Chip
SP	Stack Pointer
SR	Shift Register
TCF	Target Control Framework
TEM	Token Execution Module
TLB	Translation Lookaside Buffer
TMT	Temporal Multithreading
WB	Wishbone
WCET	Worst Case Execution Time
WRRA	Weighted Round-Robin Arbiter

1 Introduction

1.1 Motivation

With the continuous advancement of the semiconductor manufacturing technology, more and more transistors can be integrated into a single chip on the one hand. On the other hand, however, exponentially increased mask costs make it impossible to produce a small quantity of chips for an application-specific design. Consequently, merely general-purpose processors that are capable of running various applications are allowed to be mass-produced. Most of the time, such a processor can only achieve a suboptimal performance when running an application in comparison with a dedicated integrated circuit designed for that application. Aimed at addressing this issue, reconfigurable computing in different granularities has been proposed [114]. *Field programmable gate arrays* (FPGAs) and *coarse grained reconfigurable arrays* (CGRAs) are the major technologies that are currently adopted for this purpose.

One of the primary advantages of FPGAs is that logic resources can be reconfigured at the gate level. This fine-grained reconfigurability enables individual circuits to be realized, which accomplish required behaviors of different applications. However, this high flexibility comes with the disadvantage of the large amount of configuration information. For this reason, reconfiguring FPGAs takes a lot of time. In contrast to FPGAs, CGRAs provide the reconfigurability solely at the word level, which reduces the amount of configuration information dramatically and therefore results in a significant increase in reconfiguration efficiency.

Both of these technologies have one drawback in common. They require a major restructuring and/or rewriting of the application code. For FPGAs, this means that time-consuming code parts have to be realized using a *hardware description language* (HDL) like Verilog, while for CGRAs, some code structures have to be redesigned to fit into the structure of the underlying hardware. Often only a complete new development will reach the full potential of the adopted implementation technology. This causes not just a huge development overhead but also requires the expert-level knowledge about the corresponding technology.

The aim of our ongoing research is to provide a new processor paradigm, namely the AMIDAR class of processors [38]. AMIDAR is a general-purpose processor model that can be applied to various instruction sets and microarchitectures. This model allows a processor based on it to autonomously adapt to requirements of different applications, achieving a truly dynamic adaptivity. So far, our research has been conducted using the simulator of an AMIDAR-based Java processor. The research results show that the dynamic adaptivity can be best achieved by combining the processor with a CGRA which serves as a configurable accelerator [32]. This approach requires an adaption program to be executed in parallel with a running application, which generates configuration information for the CGRA at runtime. The significant performance increases observed when running real-world programs on the simulated processor indicate the great potential of the AMIDAR concept and have become the driving force that leads the research into the next phase: hardware implementation.

1.2 Research Goals

The key goal of this thesis is to develop an AMIDAR-based Java processor in the form of a soft *intellectual property core* (IP-core) for FPGAs. In the following, this processor and its simulated counterpart are

referred to as simply the AMIDAR processor and the AMIDAR simulator. The AMIDAR processor targets the embedded domain just like all preexisting Java processors. However, its main focus is general-purpose computation rather than the improvement of real-time capability, which most preexisting Java processors attempt to address. The reason for this design decision is that the AMIDAR processor must be able to execute the adaption program that performs sophisticated analysis and scheduling algorithms on the hot spots of a running application. This program has been implemented and validated on the top of the AMIDAR simulator without concern for any constraints that could occur in a hardware processor. As a result, it is much more complex than a typical embedded software. The current version of the program consists of about hundred Java classes. Depending on the complexity of the running application, it could cause a large runtime memory footprint. Therefore, to support a broad variety of applications and adaption algorithms, the AMIDAR processor should satisfy the following functional requirements:

- It can store Java classes efficiently.
- It can execute multiple threads correctly.
- It can perform garbage collection effectively.

To determine if and how well the requirements above are met, the resulting hardware implementation needs to be tested and evaluated using a standard *Java virtual machine* (JVM) benchmark suite. Besides fulfilling the fundamental requirements, this thesis also aims to implement several enhanced features that increase the performance and usability of the AMIDAR processor, including:

- *An efficient object cache*

  Java is an object-oriented language. Thus, almost all operations need to be performed on objects. Since a huge number of objects can be created by running a Java application, objects typically reside in the external memory. Exploiting an object cache will avoid the high access latency introduced by the external memory and also benefit from the object-based memory access model of Java, increasing the performance of the entire system.

- *Hardware-based system services*

  In a classical JVM, system services such as garbage collection and thread scheduling must share the processor with the running application, which causes performance overhead and additional memory usage. Employing dedicated hardware modules, these services can be performed concurrently with the execution of the application. This kind of parallelism is one of the key benefits that FPGAs provide and thus should be utilized.

- *Built-in debugging support*

  Debugging is a major challenge in developing a hardware system. Hence, one of the main goals of the AMIDAR processor is to enable and simplify debugging. Many modern FPGAs allow on-chip data to be read out at runtime, which can be exploited to realize a fine-grained hardware debugger.

1.3 Thesis Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 provides the background information on the AMIDAR concept, the Java programming language, the Java runtime system and hardware architectures proposed for scheduling threads. Chapter 3 presents the related work, including different Java processors, object caches, hardware-based garbage collectors and thread schedulers as well as several hardware debuggers. In Chapter 4, the implementation of the AMIDAR processor is described in detail. This chapter is centered around the executable format designed for the AMIDAR processor, the bytecode execution and the realization of the enhanced features mentioned above. Then, with regard to performance and size, the AMIDAR processor is evaluated in Chapter 5. Finally, Chapter 6 presents conclusions and an outlook onto future work.

2 Technical Background

2.1 AMIDAR

2.1.1 Overview

AMIDAR [38] is a general-purpose processor model for tackling today's and tomorrow's problems in the field of embedded systems. It has already been the host of some interesting research, including object oriented microarchitecture, *synthilation* [43] and CGRA-based *online synthesis* [32]. This model contains four key parts: a *token machine*, a *token distribution network*, a *data bus* and several specific functional units (FUs) such as integer ALU or heap manager. Each FU has at most one output port and an arbitrary number of input ports. Data is passed between the FUs over the data bus, as shown in Figure 2.1.

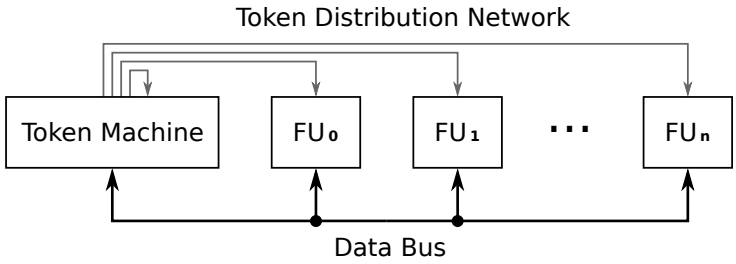


Figure 2.1: General model

The centerpiece of the AMIDAR model is the token machine that must be included in every AMIDAR implementation. It fetches instructions and decodes them into tokens for the FUs. A token can be considered as a microinstruction that needs to be executed by some specific FU. It is assigned an identifier called tag that helps determine if all operands required by the token have arrived at the data input ports of the FU. Tokens are sent over the token distribution network to different FUs. An FU will start executing an incoming token as soon as all necessary operands with the same tag as that of the token have been delivered to it over the data bus. Tokens that do not require input data can be executed immediately. After the operation has been completed, its result is transferred to a waiting FU and used as an operand for the current token of this FU. To ensure a correct operand match, the result must be assigned the identical tag as that of the token of the waiting FU.

A token can be formally defined as a 5-tuple: $T = \{UID, OP, TAG, DP, INC\}$. UID identifies which FU should execute this token. OP specifics the concrete operation. TAG serves as the identifier of the token and enables a precise operand match as mentioned above. DP describes the destination address of the result of the token. It contains the UID and a port number of the destination FU. INC is a flag and controls the generation of the tag of the result. If it is asserted, the result is tagged using $TAG + 1$; otherwise, TAG.

One of the major advantages of the AMIDAR model is that it supports simultaneous execution of instructions automatically, because their tokens can be executed on various FUs in parallel. These tokens can be clearly separated from each other by simply assigning them different tags. Also, this model allows integration of new FUs and instructions using these FUs into an existing AMIDAR processor. To meet this goal, only a small part of the token machine of the processor needs to be reconfigurable to

support inserting the token sets of the new instructions and attaching the new FUs to the token machine. Furthermore, the AMIDAR model decouples different FUs of a processor by exploiting a central bus. Each FU can be optimized separately or be customized to the end user's usage patterns, without having to consider how the other FUs are implemented. Because of this, different FUs might even be driven by various clock signals. This provides maximum design flexibility to a developer.

2.1.2 ADLA

As mentioned above, AMIDAR is a general-purpose processor model that can be applied to various *instruction set architectures* (ISAs). Since designing a token set for each instruction of a target ISA is time-consuming and error-prone, the *abstract description language for AMIDAR processors* (ADLA) and the associated compiler were developed to assist with the design process. This language abstracts away unnecessary low-level details and allows a designer to focus on the syntax and semantics of every instruction. Its compiler converts the ADLA description of an instruction set to a binary representation. Below, ADLA is briefly described based on a simple example.

ADLA Description of Java Bytecode `iadd`

This bytecode adds two 32-bit integers on the top of the operand stack and then pushes the result back onto the stack (for more details about the Java memory model, see Section 2.3.1). As Listing 1 illustrates, the ADLA description of an instruction begins with the mnemonic of the instruction (line 0). The operations that need to be performed by different FUs upon occurrence of the instruction are defined in the following curly braces by means of tokens (line 2-5).

Listing 1: Token set of iadd

```
0: iadd
1: {
2:   T(framestack, POP32, ialu.1),
3:   T(framestack, POP32, ialu.0),
4:   T(ialu,       IADD,  framestack.0)++;
5:   T(framestack, PUSH32)
6: }
```

The syntax of token definition in ADLA can be formally described by using *Backus-Naur form* (BNF) as follows:

$$token ::= T\ (FU_{exe},\ operation\ [,\ FU_{dest}.port])\ [++]$$

Identifier T indicates the beginning of a token definition. FU_{exe} and $operation$ are necessary parts of the definition, which determine the FU executing the token and the concrete operation. FU_{dest} and $port$ are optional and required only if $operation$ has a result. FU_{dest} corresponds to the FU that the result is sent to and $port$ defines the data input port of FU_{dest} that receives the result. If the tag of the result needs to be incremented, the token ends with ++ in addition.

In the example above, the first two tokens are executed by an FU called frame stack that manages the operand stack of each Java thread. Both of them perform the same operation, namely popping

the 32-bit top value from the operand stack of the current thread and sending it to the integer-ALU. The only difference between them is the port of the integer-ALU adopted to receive their results. Since *POP*32 does not require any operand, these two tokens may have the same tag without causing any operand mismatch. Their results and the third token are also assigned this tag so that the integer-ALU can determine whether both operands of IADD have arrived. The result of IADD is returned to the frame stack and then used as the operand of the last token. To indicate that this operand belongs to the last token rather than the first two that are also executed by the frame stack, a different tag needs to be assigned to it. This is achieved by inserting ++ at the end of the third token. Accordingly, the tag used by the last token also needs to be incremented to guarantee a correct operand match. For this purpose, the last token is separated from the third one by using a semicolon. In contrast, tokens with the same tag are separated by commas, like the former three ones in this example.

Compilation of ADLA Description

After all instructions of a target architecture have been described by using ADLA, an equivalent in-memory representation is generated by the ADLA compiler automatically. This representation consists of two parts: a *meta-table* and a *token-matrix*. The former keeps the fundamental information about the instructions and the latter saves their token sets.

The meta-table contains a single entry for each instruction, which is indexed by the opcode of the instruction. For example, in the context of Java, the meta-information of **`iadd`** is stored in the 96th entry because the opcode of **`iadd`** is equal to 96. The meta-information of an instruction includes:

- The number of its parameters.
- A flag indicating if the instruction performs a jump operation.
- The number of rows used to save its tokens in the token matrix.
- The offset of its token set inside the token matrix.

Each column of the token-matrix corresponds to an FU and each row contains the tokens of a token set, which can be delivered to different FUs with the same tag. This implies that multiple tokens which are defined sequentially in a token set can be sent to the corresponding FUs concurrently. However, a single row does not always include a token for every FU. Therefore, each cell of the matrix has a flag indicating whether it holds a valid token entry or not. Also, there is an additional flag for every row. If this flag is asserted, the tag of the tokens held in the next row needs to be incremented; otherwise, the tag remains unchanged.

Assume that some customized version of the AMIDAR processor is solely composed of a token machine (TM), a frame stack (FS), a heap manager (HM) as well as an integer-ALU (IALU). Figure 2.2 illustrates a snapshot of the meta-table and token-matrix generated for this processor. To simplify the representation, this snapshot only shows the tokens defined for **`iadd`**. As can be seen in the meta-table, this bytecode neither has any parameter nor executes a jump operation. Its tokens are stored in a total of three rows in the token-matrix (row 384-386).

Row 384 of the token-matrix contains two tokens of **`iadd`** because they have the same tag and are distributed to various FUs. Although the second token executed by the frame stack, which is stored in

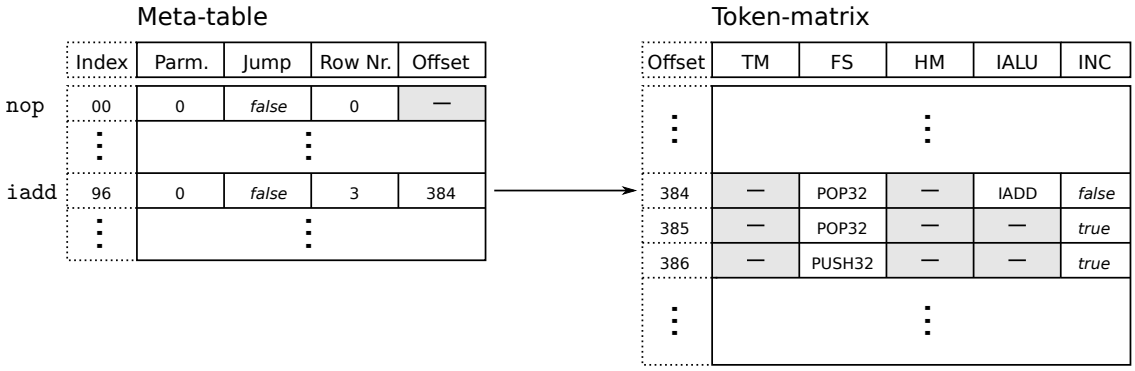


Figure 2.2: Meta-table and token-matrix

row 385 also has this tag, it cannot be sent with the first one together, since the token machine can only deliver one token to each FU at a time. The tag of the fourth token in row 386 is incremented by asserting the INC-flag of row 385 to ensure a correct operand match as described above. Note that the INC-flag of the last row of a token set is always asserted to clearly separate the current token set from the next one in the token-matrix.

2.2 Java

2.2.1 Java in Embedded Systems

Java is one of the most popular programming languages in the world. Its usage spans a broad range of areas, from the development of server-side software to the creation of Android applications. Even in the field of embedded systems, where C is traditionally considered as the dominant language, Java is becoming increasingly attractive for the following reasons:

- Java is a simple object-oriented language. Peripherals and sensors can be represented in a straightforward manner by means of objects.
- Java programs are highly portable. Classes can be shared among different devices without the need to recompile them.
- Java is equipped with a rich set of libraries. Exploiting the *application programming interfaces* (APIs) provided by these libraries, development productivity can be greatly increased.
- Java supports multi-threading at the language level. Parallel activities of various peripherals can be easily modeled with threads.
- Java is safer and more reliable than C. Safety from the beginning has been one of the key goals of Java. To meet this goal, Java provides multiple mechanisms. First, it performs strong type checking at both compile time and runtime. Second, objects are accessed through references instead of error-prone pointers. Third, memory management in Java is fully automatic, including object allocation, initialization and reclamation. Also, the built-in exception handling reports errors explicitly, which simplifies debugging significantly.

- There is a worldwide population of Java developers that have trained in different domains. Their programming skills and experience can be utilized in the field of embedded systems directly.

To satisfy requirements from different application areas, various *Java runtime environments* (JREs) are available. For example, *Java Platform, Standard Edition* (Java SE) targets desktop and server class computers, while *Java Platform, Micro Edition* (Java ME) is aimed at headless embedded systems on devices with one megabyte or less of memory. From Java version 1.2 to 1.4, these JREs are named as *Java 2 Standard Edition* (J2SE) and *Java 2 Micro Edition* (J2ME) respectively. J2ME includes two further configurations defined to classify embedded systems at a fine-grained level, namely *Connected Device Configuration* (CDC) and *Connected Limited Device Configuration* (CLDC). Such configurations are actually the specifications for the Java runtime systems that a J2ME device must support. CLDC was designed for devices with 160 KB to 512 KB total memory and has been chosen as the specification for the vast majority of preexisting Java processors.

2.2.2 Concurrency

Java is an explicitly multi-threaded language. This section introduces the thread and synchronization models of Java briefly. In addition, several relevant Java methods used to develop multi-threaded programs are also described below.

Thread Model

Java supports multi-threading at the language level directly. For creating threads and synchronizing their executions, Java provides multiple API methods. Using these methods, programmers can generate a new thread instance, set its attributes (e.g. its priority) and define the task run by it. Once a thread is started, the Java runtime system takes over responsibility for scheduling the execution of the thread regarding its attributes.

The scheduling model of Java is preemptive, which means that the runtime system assigns each thread a time-slice to execute its task, interrupts the execution after the expiration of the time-slice and context switches to another thread. This process is repeated periodically until all threads have finished their tasks. In this way, different tasks can be performed simultaneously (more precisely, pseudo-simultaneously), which provides two major advantages for program design and development.

The first advantage is speeding up the execution of multiple tasks on a single processor system. This may sound a little counterintuitive at first, especially when considering the overhead caused by context switches. However, a key point is that an I/O operation often takes much more time than a context switch. Without using multiple threads, the program blocks during the entire I/O process. In contrast, it can execute further, if other threads are available, which perform nonblocking tasks.

Another advantage as a direct consequence of the first one is the improvement of the responsiveness of a program. As mentioned, the AMIDAR processor is intended to be used in the field of embedded systems. Such a system typically includes a set of peripherals that operate at different speeds for various purposes. If the system needs to check the status of each of its peripherals in a round-robin fashion and then executes some specific operation on the corresponding peripheral according to the check result, it cannot respond to a request from any of other peripherals before the current operation is complete.

In general, the request of an external device is sent to a processor in the form of an interrupt. Using a dedicated *interrupt service thread* (IST) for each peripheral, the system can handle requests from external devices as desired, achieving the maximum responsiveness. To meet this goal, the interrupt handling model of the AMIDAR processor has been integrated into the thread model of Java completely.

Synchronization Model

Since multiple threads may share common data or resources, Java employs a synchronization mechanism known as *monitor* for the purpose of thread-safety. Every object is associated with a single monitor that ensures the mutually exclusive access to the object to prevent collisions over common resources. Java provides built-in support for monitor in terms of the **synchronized** keyword that can be used on both methods and instruction sequences called critical sections. At the source code level, there is no difference between a synchronized method and a critical section from the aspect of semantics. For example, the codes shown in Listing 2 and 3 perform exactly the same operation. However, the Java compiler treats them differently, which is illustrated in Listing 4 and 5.

Listing 2: Synchronized method

```
private int cnt = 0;

public synchronized void inc() {
    cnt++;
}
```

Listing 3: Critical section

```
private int cnt = 0;

public void inc() {
    synchronized(this) {cnt++;}
}
```

Listing 4: Bytecodes of synchronized method

```
// 0: aload_0
// 1: dup
// 2: astore_1
// 3: monitorenter

/* cnt++; */
0: aload_0
1: dup
2: getfield      #12
5: iconst_1
6: iadd
7: putfield      #12

//14: aload_1
//15: monitorexit
```

Listing 5: Bytecodes of critical section

```
 0: aload_0
 1: dup
 2: astore_1
 3: monitorenter

 /* cnt++; */
 4: aload_0
 5: dup
 6: getfield      #12
 9: iconst_1
10: iadd
11: putfield      #12

14: aload_1
15: monitorexit
```

As can be seen in Listing 5, the Java compiler explicitly inserts two synchronization-specific bytecodes, namely **monitorenter** and **monitorexit** that enclose the critical section as well as several bytecodes (e.g. line 0-3) that compute the operands for them. Since a synchronized method lacks

monitorenter and **monitorexit** in its bytecode stream, the Java runtime system has to check its access flag additionally to determine how to execute it properly.

If a thread needs to execute a synchronized method or a critical section on an object, it must acquire (or enter if the monitor is considered as a door to the object) the monitor of the object at first. Otherwise, it must block until the monitor is released. Once the monitor of an object has been owned by some thread, its owner may reenter it recursively. The runtime system needs to track how many times the monitor has been entered, using a counter. Each time the owner leaves the monitor on return from a synchronized method or leaving the scope of a critical section, the internal counter of the monitor is decremented by one. Only if the counter of the monitor reaches zero, the monitor may be released and allows to be entered by another thread. If only one thread is blocked by the released monitor, it may acquire the monitor directly. However, if multiple threads are blocked, one of them needs to be selected and assigned the monitor by the runtime system. The selection algorithm is not explicitly defined in the Java specification and therefore is implementation-dependent.

Thread-specific Methods

As mentioned above, a number of methods are available for writing multi-threaded programs. They can be categorized into two groups: thread-specific and synchronization-specific methods. The former group contains all methods declared in class **java.lang.Thread**, some of which are native methods and others are implemented based on the native ones. The latter group includes several native methods declared in class **java.lang.Object**. This subsection provides an overview on the thread-specific methods, while the synchronization-specific methods are presented in the following subsection.

constructor: A new thread instance can be created by using the constructor of class **Thread**. The task that needs to be executed by the thread can be optionally passed in as an argument of the constructor. In this case, the task object must be an instance of some class that implements interface **Runnable**. This object is then assigned to a field of the thread instance, which is called **target**.

run: This method defines the code sequence that performs the actual task. If the **target** field is not null, it just invokes the **run**-method on **target** as follows:

Listing 6: Thread.run()

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

An alternative approach to defining a task is to override the **run**-method in a subclass of **Thread** and create new thread instances from the subclass instead of **Thread**.

start: A newly created thread instance is not taken into account by the runtime system for scheduling until its **start**-method has been invoked. Through the invocation of this method, the thread is attached to some internal data structure of the runtime system (e.g. a priority queue), which holds all ready threads. During the next scheduling process, the runtime system will select one of these ready threads to replace the currently running thread, using an implementation-specific algorithm.

setPriority: Using this method, a thread may be given an explicit priority; if not, it simply inherits the priority from the thread creating it, i.e. its parent thread. Java defines a total of 10 priority levels from 1 to 10, where 1 represents the minimum priority and 10 the maximum. Threads with higher priority are executed in preference to threads with lower priority.

yield: This method gives the runtime system a hint that the thread calling it is willing to give up the processor in order to allow another ready thread to be executed. Nonetheless, the runtime system may choose to ignore the hint and let the current thread run further. This means that the result of the execution of this method is implementation-specific and therefore unpredictable.

sleep: The invocation of this method causes the calling thread to sleep for a given time. During this time, the execution of the thread may not be resumed. A very important thing to note is that the monitors owned by the calling thread are not released and thus can not be entered by other threads.

join: If a running thread calls this method on another thread, *T*, it cannot proceed before *T* has terminated. Optionally, a timeout value may be given to limit the joining duration. If the timeout value expires before *T* has terminated, the calling thread becomes ready again. Like the **`sleep`**-method, this method does not release the monitors owned by the calling thread.

Synchronization-specific Methods

All synchronization-specific methods are based on the monitor construct of Java. Class **`Object`** provides three native methods that utilize the monitor construct to support more sophisticated synchronization mechanisms among threads. Invoking these methods on an object has one common constraint that the calling thread must already own the monitor of the object, otherwise an exception will be thrown by the runtime system.

wait: The invocation of the **`wait`**-method on an object causes the calling thread to give up the monitor of the object and start waiting on the object (i.e. the thread is suspended). A waiting thread is not considered by the runtime system during the scheduling process. The released monitor is assigned to one of the blocked threads requiring this monitor, based on an implementation-specific selection algorithm. Optionally, a timeout value may be given as an argument to limit the waiting duration. After the expiration of the timeout value, the waiting thread becomes ready automatically, i.e. it is allowed to be scheduled to run again. If no timeout value is given, the thread waits until either of the following methods is invoked.

notify: This method notifies the runtime system that the monitor of the object on which the method has been invoked is about to be released. The runtime system wakes up one of the threads waiting for the object's monitor according to some implementation-dependent algorithm. However, the awakened thread must enter the monitor first before it can run further.

> Java Specification [49]: *The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.*

Note that the internal counter of the monitor needs to be reset to the status before calling the **`wait`**-method once the awakened thread reenters the monitor.

notifyAll: Calling this method on an object wakes up all threads waiting for the monitor of the object instead of a single one, which is the only difference between the **notify**- and **notifyAll**-methods. All awakened threads have to compete with any other threads that are also trying to enter the monitor. Three important things to note about both of the **notify**- and **notifyAll**-methods are that:

1. If there is no waiting thread, nothing will happen through executing these methods.
2. The monitor is actually not released after calling either of these methods until the thread returns from the synchronized-method or leaves the critical section, or executes the **wait**-method on the corresponding object.
3. Unlike calling the **wait**-method, the thread that calls either of these methods proceeds until the next thread context switch, otherwise it would not be able to release the monitor indeed.

For the latter two reasons above, the **notify**- and **notifyAll**-methods should always be invoked at the end of a synchronized method or a critical section, or just before calling the **wait**-method so that the awakened thread or threads may really acquire the monitor.

2.3 Java Runtime System

2.3.1 Memory Model

The ISA defined by the Java bytecode partitions memory into three runtime data areas: *method area*, *heap* and *Java stack*. In the following, each of these data areas is described briefly.

Method Area

The method area can be considered as the code memory of a Java runtime system and is shared among all threads of an application. It holds the meta-information about the loaded classes, the bytecode streams of the methods defined in these classes as well as a set of constant values. All these information and data are generated by the Java compiler and saved statically in individual class files, using a platform-independent format. At runtime, they need to be extracted from the class files and loaded into the method area so that they can be accessed by the runtime system. Their representation inside the method area is implementation-specific and should be designed to facilitate efficient execution of the application.

Heap

Like the method area, the heap is also shared among all threads. It manages class instances and arrays created at runtime. Since Java does not allow an object to be deallocated explicitly, the runtime system needs to provide a garbage collector that reclaims the memory of unreachable objects. The manner in which the garbage collector cleans up the heap is not explicitly defined and therefore can be designed based on the specific characteristics of the runtime system.

Java Stack

In contrast to both data areas above, every thread is assigned a dedicated Java stack as it is created. Each time a thread invokes a method, a new frame is pushed onto the thread's Java stack. The

method uses this frame to store its parameters, local variables, intermediate computation results and other context-related data. Once the method invocation is complete, the frame is discarded.

The method that is currently being executed by a thread is referred to as the thread's *current method*, and its frame is known as the *current frame*. The runtime system executes bytecodes solely on the current frame. A frame is no longer current if its method invokes another method or its method completes. In the former case, the frame of the invoked method becomes the current frame. In the latter case, the current frame is removed from the Java stack and the previous frame becomes the current one.

The stack frame of a method consists of three parts, namely a local variable array, an operand stack and a frame data section, as explained below.

Local Variable Array: All local variables of the method are saved in an array of words. The length of this array is determined at compile time and is loaded together with the bytecode stream of the method into the method area at runtime. An element of this array can be addressed by its index directly. A value of type **`long`** or **`double`** occupies two successive entries in the array, while a value of any other type only one. This implies that values of type **`byte`**, **`char`** and **`short`** need to be cast to integers before they are stored into the local variable array. Besides holding the local variables, this array is also employed to pass the parameters of the method. Any parameters of the method must be placed into the successive local variables starting from local variable 0, in their declaration order.

Operand Stack: The operand stack is a 32-bit *last-in-first-out* (LIFO) stack that serves as the primary work space of the Java runtime system. Java supplies a number of load and store bytecodes that are solely used to transfer values between the operand stack and other data areas, including the method area, the heap and the local variable array. The vast majority of the remaining bytecodes take values from the operand stack, perform corresponding operations on them, and then push the results back onto the operand stack. For this reason, the depth of the operand stack varies continuously as bytecodes are executed. Pushing a value of type **`long`** or **`double`** onto the operand stack increases its depth by two, while pushing a value of any other type increases its depth by one. Also, the operand stack is used to prepare parameters for a method and receive the method's result. The parameters must be pushed onto the operand stack in the order in which they are declared.

Frame Data Section: The frame data section is intended to assist the runtime system with constant pool resolution, method return and exception dispatch. However, its actual layout, size and functions are implementation-dependent and therefore can be quite different from one runtime system to another. For example, in a classical JVM, it might store a reference to the constant pool of the class that defines the method, the value of the *program counter* (PC) of the calling method (i.e. the caller) as well as a reference to the exception handler table of the method. In contrast, it solely saves the context data of the caller in the AMIDAR processor.

2.3.2 Object Access

Object Addressing

As described in Section 2.3.1, the heap is the runtime data area from which objects and arrays are allocated. While developing a heap management system, a key design decision that needs to be made is the way how objects are addressed. Two important schemes include *direct addressing* and *logical addressing* (or *indirect addressing*).

In the former scheme, an object is referenced by its base physical address or its base virtual address if a virtual memory system is used. This allows the physical or virtual address of a field of the object to be calculated by simply summing the object's base address and the field's offset. However, this scheme unduly complicates object relocation which is necessary for heap compaction. The reason is that all references to an object must be updated across the whole memory system from the frame stack to the heap at a time after the object has been reallocated.

In contrast, an object is referenced by a location-independent identifier in the latter scheme, which is referred to as *handle* in this thesis. A handle is actually an index into a table called *handle table* below. The handle table holds meta-information about every object, like its base physical address. This scheme greatly simplifies object reallocation because an object's memory address only needs to be updated in the handle table once. The primary drawback of this scheme is the indirection overhead when accessing a field of an object, since the object's memory address is not directly available and must be retrieved from the handle table. This issue can be overcome by using an object cache, because the handle table look up is only necessary if a cache miss occurs.

Object Caching

A Java program can create a large number of objects throughout its lifetime. Therefore, the heap is usually resident in external memory to provide sufficient storage space. To avoid the high access latency introduced by the external memory, an object cache is typically employed, whose architecture is determined by the object addressing scheme used, as discussed below.

In the case of the direct addressing scheme, a physically addressed cache is the only choice, if no virtual memory system exists. Otherwise, either a physically or a virtually addressed cache can be used, depending on where the *translation lookaside buffer* (TLB) is placed, before or after the cache. Both of these caches treat data blocks of the external memory as their first-class entities. Thus, every word held in a cache line is accessible, providing the full cache capacity to the runtime system. Also, this allows for prefetching an adjacent object on the occurrence of a cache eviction.

Upon an access to a field of some object, both of these caches take the sum of the object's base address and the field's offset, namely the address of the field, as input. Each of the tag value, the cache line index and the cache line offset is just a subset of the field's address bits. In this way, successive fields of the object can be distributed through the entire cache, reducing conflicts. The major difference between a physically and virtually addressed cache is that the latter one needs to handle the issue of address aliasing, while the former one does not.

For the logical addressing scheme, a physically addressed cache could be used. In this case, the handle table would serve as the TLB. However, this would incur a delay of at least one clock cycle due to the handle table lookup. A classical way to solve this issue is to employ a logically addressed cache whose cache lines are tagged directly with handle-offset pairs. Although the basic idea of a logically addressed cache is quite similar to that of a virtually addressed one, they differ from each other in several aspects.

Unlike a virtual address, the handle of an object is the object's unique identifier. Therefore, a logically addressed cache does not need to deal with address aliasing and can be implemented as a *logically-indexed*, *logically-tagged* cache (LILT). In such a cache, the first-class entities are objects rather than memory blocks. This means that each cache line can only be associated with a single object. As

a result, some words at the end of a cache line will not be used, if the size of the cached object is not an integral multiple of the cache line size. This phenomenon is referred to as *external fragmentation* below. The direct consequence caused by the external fragmentation is the reduction of the effective cache capacity. Another negative effect is the increased complexity of the write-back logic, because only the valid part of a cache line may be written back to the heap. The external fragmentation can be considered as the price paid for the major benefit brought by the indirect addressing scheme, namely the easy implementation of heap compaction.

A logically addressed cache does not support prefetching an adjacent object upon a cache miss. However, the vast majority of objects are short-lived [14, 127] and will die in the cache [123], which means that prefetching would be actually only important for long-lived objects that need to be cached repeatedly. According to previous research, there is little spatial locality between long-lived objects [13]. Consequently, the lack of object prefetching should not result in a notable performance loss.

Another key difference between a logically and virtually addressed cache is the way how cache index is generated. Upon an access to a field of an object, a logically addressed cache typically generates the cache index with several handle bits and several offset bits in the hope of reducing intra- and inter-object conflicts at the same time. If the index solely consisted of handle bits, the fields of a large object would be stored in a single cache set, leading to increased intra-object conflicts. If the index was only made up of offset bits, small objects would be restricted to the cache sets at the beginning of the cache, increasing inter-object conflicts. Which and how many bits should be selected from each of the handle and the offset for the purpose of index generation are implementation dependent issues. In Section 3.2 below, several index generation schemes are discussed in detail. Since only a part of the offset bits are used to calculate the cache index, the maximum cache space that may be occupied by an object is limited. In contrast, a virtually addressed cache does not have such a limitation and therefore can even be filled with one single object.

2.3.3 Garbage Collection

As mentioned in Section 2.3.1, when running a Java program, objects allocated from the heap cannot be explicitly deallocated by the program. To avoid running out of memory, a runtime system must provide a mechanism to automatically clean up the heap, which is known as garbage collection [69]. The part of the runtime system that is employed to perform garbage collection is typically referred to as garbage collector. A garbage collector has two major tasks: detection of garbage objects and reclaiming the memory occupied by such objects. An object is considered garbage, if it is no longer referenced by the program, otherwise it is said to be live. In the following, the implementation of a garbage collector is discussed from different points of view briefly.

Garbage Detection

There are two well-known approaches to distinguishing between live and garbage objects, namely *reference counting* [27] and *tracing* [69]. A reference counting garbage collector keeps track of the number of references to every object. Once the reference number of an object becomes zero, the object can be garbage collected. This approach allows any unreferenced object to be detected and removed on the fly, which makes it suitable for real-time systems in particular. Its main disadvantage is that it cannot

recognize reference cycles. Also, maintaining a reference count for each object on the heap increases both performance overhead and memory usage. Due to these drawbacks, the vast majority of modern garbage collectors are based on the latter approach, namely tracing. Therefore, the discussion below is centered around tracing garbage collection.

A tracing garbage collector determines which objects are still referenced by the program, i.e. it detects live objects rather than garbage objects. For this purpose, it traces out the graph of references starting from a set of root objects and marks every reachable object as live. Accordingly, objects that are not reachable from the root set will remain unmarked and become eligible for garbage collection.

An object is considered a root object if it is directly accessible to the program. Thus, although its definition is implementation-dependent, the root set should always include any object references stored on any Java stack and in the static fields of any class. An object referenced by a root object is reachable and therefore is a live object. An object referenced by a live object is reachable as well, which means that object reachability is a transitive closure. All reachable objects can be potentially accessed by the program and thus may not be removed.

Inside a runtime system, especially on the Java stack and the heap, an object reference is represented in the same way as a 32-bit primitive value [50]. If a garbage collector can tell apart a reference from a primitive value so that the references to live objects can be precisely identified during the tracing process, it is called a *precise* collector, otherwise a *conservative* collector. A conservative collector cannot recognize a garbage object, if that object happens to be pointed to by some primitive value which appears to be a valid reference. As a result, garbage collection has to be performed more frequently. Implementing a precise collector requires assistance from both of the runtime system and the compiler. The runtime system must be able to extract the references stored on any Java stack to construct the root set. The compiler needs to generate type information about every field of a class to allow each reference contained in an object of the class to be exactly traced.

In practice, a tracing garbage collector can be realized by using the *tri-color* algorithm [31]. This algorithm adopts three different colors to indicate the state of an object, namely *white*, *gray* and *black*. Initially, all objects are white except the root ones, which are marked gray. After all white objects referenced by a gray object, namely O, have been marked gray, the color of O is changed to black. This process repeats itself until there is no gray object anymore. Then, any objects that are still white can be garbage collected. Several other algorithms [18, 102, 128] are also based on the tri-color marking abstraction, but they do not assign a color to an object explicitly. Instead, they exploit a stack to keep the references to gray objects during a trace. Also, a single-bit flag is associated with each object to denote whether the object has ever been on the stack. At the beginning of a new trace, the stack is initialized by pushing the references contained in the root set onto it. Upon pushing each reference, the flag of the corresponding object is asserted. After that, the references on the stack are popped one by one. If the object that a popped reference points to contains references to some other objects with unset flags, these references are pushed onto the stack. The trace completes once the stack becomes empty. Objects whose flags stay unset are considered white and can be removed.

Memory Reclamation

When an object is no longer referenced by the program, the memory that it occupies needs to be reclaimed and made available again for subsequent new objects. This goal can be achieved in different ways. A classical *mark-sweep* garbage collector [69] maintains a linked list of available memory blocks (i.e. a free list) and performs collection in two phases, namely a mark phase and a sweep phase. All live objects are identified and marked in the former phase. In the latter phase, the entire heap is swept (i.e. every object on the heap is checked) and the memory block occupied by any unmarked object is appended to the free list. To allocate a new object, the runtime system needs to look up a memory block in the list into which the object will fit. The major problem of this approach is heap fragmentation which could cause the runtime system to run out of memory unnecessarily. To avoid this problem, either of the following approaches can be utilized: *copying* [33] and *compacting* [57].

In a copying garbage collector, the heap is divided into two equally sized semi-spaces. Only one of these spaces is used between two successive garage collection cycles, while the other space simply stays inactive. Once the active space fills up, the program execution is suspended and the garbage collector starts traversing the graph of references from the root set. Live objects are copied from the active space into the inactive one as they are encountered during the traverse. These objects are placed side by side in the inactive space, eliminating memory fragments between them. After all live objects have been reallocated, the roles of the two spaces are flipped, with the current inactive space becoming the new active space. Then, the program execution resumes. The primary drawback of the copying approach is that only half of the available memory can be used at any time. Also, long-lived objects will be copied between the two spaces in every garbage collection cycle. A generational collector addresses the latter issue by grouping objects by age and garbage collecting younger objects more often than older ones. In such a garbage collector, the heap is partitioned into multiple sections. Each of these sections serves one *generation* of objects and can be cleaned up using a copying garbage collector. Since the vast majority of objects are short-lived, only a small fraction of young objects will survive their first garbage collection cycle. After an object has survived a few garbage collection cycles, it is considered mature and moved to the next older generation. Every older generation is garbage collected less often than the next younger generation. In this way, the efficiency of the underlying copying garbage collector can be greatly improved, however, at the expense of significantly increased implementation complexity.

A compacting garbage collector is typically referred to as a *mark-compact* garbage collector. This implies that it also needs to perform collection in two separate phases. In the compact phase, objects that have been marked in the previous phase are moved over free memory space toward one side of the heap (the to-side), which results in a large contiguous free memory area on the other side of the heap (the from-side). This approach allows the entire available memory to be used by the runtime system. Furthermore, long-lived objects will accumulate at the to-side of the heap, which avoids reallocating them repeatedly.

Concurrent Garbage Collection

In discussions about concurrent garbage collection, the executing program is typically referred to as the *mutator*. A concurrent garbage collection algorithm must ensure that the following two conditions will never be fulfilled at the same time [121]:

- A reference to a white object is assigned to some field of a black object.
- The white object has no other reference pointing to it.

Otherwise, the white object could be wrongly considered garbage and therefore be removed. To prevent this situation, there are two basic approaches. One adopts a *read-barrier* to change the color of a white object to gray as soon as the mutator tries to access that object. Since the mutator can never get any reference to a white object, it cannot assign such a reference to a field of a black object. This approach avoids the occurrence of the two conditions in advance. The other approach, in contrast, employs a *write-barrier* to detect the occurrence of the first condition and then eliminate either of the two conditions on the fly. Below, a brief overview on the popular concurrent garbage collection algorithms is provided, where the former three are the variants of the mark-sweep algorithm and the latter two are the variants of the copying algorithm.

- *Steele's concurrent mark-sweep collector* [102] is a stack-based tri-color marking collector using a simple write-barrier. Once the mutator attempts to create a reference from a black object to a white one, the write-barrier reverts the black object to gray.
- *Dijkstra's on-the-fly collector* [31] is a write-barrier collector that uses the tri-color marking abstraction explicitly. Its write-barrier changes the color of a white object to gray as soon as that object is referenced by a black object.
- *Yuasa's sequential collector* [128] is also a stack-based tri-color marking collector which aims to eliminate the second condition rather than the first one. For this purpose, its write-barrier marks a white object gray, if a reference to that object is overwritten by the mutator.
- *Baker's incremental copying collector* [12] is a read-barrier collector which exploits a stack to realize the tri-color marking abstraction implicitly. It flips the two semi-spaces at the beginning of a garbage collection cycle and copies every live object from the inactive space into the active one. The color of every newly copied object is changed from white to gray. After all objects referenced by a gray object have been copied into the active space, that gray object is colored black. If the mutator accesses any object in the inactive space, the object must be first copied into the active space and marked gray, which is enforced by the read-barrier.
- *Nettle's replication copying collector* [74] is a write-barrier collector which flips the two semi-spaces at the end of a garbage collection cycle. As the collector copies live objects into the inactive space, the mutator continues to access objects in the active space. For this reason, the objects in the inactive space are said to be the replicas of the corresponding objects in the active space. The write-barrier ensures that any modification made to an object in the active space will be synchronized to its replica so that the mutator sees the correct values after the flip.

As mentioned above, these algorithms are based either on the mark-sweep algorithm or on the copying algorithm. This is because both of them allow unused memory to be reclaimed on the fly, without interfering with the execution of the mutator (in particular, allocation of new objects) in a notable way. However, their major disadvantages are also inherited by their concurrent variants.

The mark-compact algorithm is considered less suitable for concurrent garbage collection due to the way how it reclaims memory. In the compact phase, a new object may only be allocated from the heap area which has already been compacted. If this area does not contain enough memory to satisfy the allocation request, the execution of the mutator has to be paused until sufficient space has been reclaimed for holding the object. The duration of the pause depends on the object's size and thus is unpredictable. The algorithm proposed in [102] adopts an additional free list to facilitate object allocation, which, however, causes that the heap fragmentation cannot be eliminated completely.

The heap compacting scheme proposed in this thesis allows a new object to be allocated in constant time, providing all benefits of the compacting approach to the runtime system. In Section 4.5.5, this scheme is presented in detail.

Finalization

In Java, every class inherits the **finalize**-method from class **java.lang.Object**. A class may override this empty method to explicitly release some non-memory resources or to perform other cleanup. The garbage collector invokes the **finalize**-method on an unreachable object prior to reclaiming the object's memory. This method can perform any operations, including resurrecting the object. However, it may only be run once throughout the lifetime of the object. Any unreachable object that has been finalized can no longer have any affect on the running program and therefore can be removed safely.

The Java specification does not define which thread should invoke the **finialize**-method, but it requires that the thread employed for that purpose may not hold any user-visible monitors. Also, if an uncaught exception is thrown by the **finalize**-method, the exception is ignored and the execution of the method terminates immediately.

Reference Objects and Reachability Levels

An object is called a *reference object*, if it is an instance of one of the following three classes: **SoftReference**, **WeakReference** and **PhantomReference**. All these classes are derived from abstract class **java.lang.ref.Reference**. A reference object can hold a special reference to another object which is known as a *referent object*. Depending on from which class the reference object has been created, the special reference held by it is referred to as a *soft reference*, a *weak reference* or a *phantom reference* respectively.

As a reference object is created, a reference to its referent object must be passed in as a constructor parameter. If the reference object is an instance of **SoftReference** or **WeakReference**, it will hold this reference as long as it is not cleared by the program or the garbage collector (a reference object is said to be cleared if the reference to its referent object is overwritten with null). If the reference object is an instance of **PhantomReference**, it may only be cleared by the program.

A phantom reference object must be associated with a *reference queue* when it is created, which is optional for a soft or weak reference object. A reference object needs to be appended by the garbage collector to its associated queue under certain circumstances. For this purpose, class **Reference** provides an enqueue method. If a reference object does not have an associated queue, nothing will happen when calling this method on it. Thus, the garbage collector can simply invoke this method on any reference

object that should be enqueued without the need to know whether the object is actually associated with a reference queue.

Beginning with Java 1.2, any reachable object may have one of four reachability levels which are described from strongest to weakest as follows:

- An object is *strongly reachable* if it can be reached from the root set without traversing any reference object. The garbage collector may do nothing to a strongly reachable object except reallocate it.
- An object is *softly reachable* if it is not strongly reachable but can be reached through one or more soft references. The garbage collector may choose to clear all soft references to that object. Any newly cleared soft reference object that is associated with a reference queue will be enqueued immediately or at some later time.
- An object is *weakly reachable* if it is neither strongly reachable nor softly reachable but can be reached through one or more weak references. The garbage collector must clear all weak references to that object. Any newly cleared weak reference object that is associated with a reference queue will be enqueued immediately or at some later time.
- An object is *phantom reachable* if it is not strongly reachable, softly reachable nor weakly reachable but can be reached through one or more phantom references, and it has been finalized. If the garbage collector encounters a phantom reference object whose referent object is phantom reachable, the phantom object will be enqueued immediately or at some later time. If some object is not phantom reachable only because it has not been finalized yet, the garbage collector is allowed to finalize it.

2.3.4 Thread Scheduling

This section first provides an overview on the major thread scheduling algorithms. Then, the classical priority inversion problem is described, as well as the solutions of avoiding it.

Thread Scheduling Algorithms

Thread scheduling algorithms are designed to optimize various metrics. The most important metrics for embedded systems include:

- ***Processor utilization***: the fraction of processor cycles that are actually employed to execute user tasks. The rest of processor cycles can be considered as overhead caused by system tasks like context switching.
- ***Interrupt latency***: the interval from the arrival of an interrupt until the corresponding IST starts running.
- ***Enforcing priorities***: the ability to ensure that the priorities of threads are reflected by the scheduling decisions clearly.
- ***Fairness***: the ability to avoid starvation of any thread.

- ***WCET-predictability***: the ability to determine the worst case execution time.
- ***Deadline missing rate***: the percentage of deadlines that are not met.

The former four are the common metrics while the latter two are typically used to evaluate the quality of real-time systems. A key thing to note is that these metrics may interfere with each other and therefore cannot reach their theoretical optimum at the same time. For example, a purely priority-based scheduling algorithm may cause that lower-priority threads will never be scheduled, if a higher-priority thread runs continuously, which results in poor fairness. Due to these mutual influences between different metrics, a scheduling algorithm should be designed under consideration of the specific requirements of the target system.

Generally, the workload of an embedded system can be represented as a task set containing both periodic and aperiodic tasks. A thread that executes a periodic task needs to be assigned time-slices in a constant rate to ensure the fairness, whereas a thread that executes an aperiodic task like interrupt handling should be started as soon as possible to achieve the minimum response latency. Without concern for priority, the most fundamental scheduling algorithms include *round-robin* (RR) and *first-in-first-out* (FIFO) [101]. The former schedules all ready threads in a fixed order and assigns each of them a time-slice periodically. This algorithm fits the preemptive scheduling model in its nature and therefore is suitable for scheduling periodic tasks. The FIFO algorithm is typically used by the non-preemptive scheduling model or just by the RR algorithm to determine the scheduling order.

For the preemptive, priority-based scheduling model used by Java, the most accepted algorithm is *multilevel queue* (MLQ) [1] that partitions all ready threads into separate queues according to their priorities. The threads at the same priority level (i.e. in the same queue) are scheduled by using the RR algorithm [101]. Besides the intra-queue scheduling, this algorithm also needs to make scheduling decisions among different queues (i.e. the inter-queue scheduling), depending on the application-specific requirements. For example, a real-time system may prefer purely priority-based scheduling so that only the threads at the highest priority level are taken into account, which could cause starvation of lower-priority threads. Another common solution is to adopt the RR algorithm for both of the intra- and inter-queue scheduling. In this case, after each of the threads at a priority level has been assigned a time-slice, the threads with the next lower priority may be scheduled to run. The priority of each thread is reflected by the length of its time-slice: the higher the priority, the longer the time-slice.

The *bitmap* scheduling algorithm is a simplified version of MLQ with the constraint that each priority level may only contain a single thread. Another variation of MLQ is the *multilevel feedback queue* (MLFQ) algorithm [101] that assigns every newly created thread the highest priority and demotes it to the next lower-priority queue each time after the expiration of its current time-slice. As a result, newer and shorter threads are favored over older and longer ones.

In the real-time theory, the two most often studied and employed algorithms are *rate monotonic* (RM) and *earliest deadline first* (EDF) [67]. Both of these algorithms can be implemented based on MLQ. The only difference between them is the way how they determine the priority of a thread. RM requires that each thread must be assigned a static priority inversely proportional to its period before it is started. During the entire lifetime of the thread, the priority remains frozen. In contrast to RM, EDF updates the priorities of threads dynamically according to their absolute deadlines. The absolute deadline of each thread is calculated on the fly, using the current system time and the relative deadline of the thread.

The *least slack time first* (LSTF) algorithm is an extension of EDF, which uses the difference between the absolute deadline and the remaining execution time of a thread as the priority of the thread.

Although previous research [23] shows that an EDF-based scheduler provides a lower preemption rate and therefore allows for better processor utilization, most commercial *real-time operating systems* (RTOS) use a RM-based scheduler for several reasons [101]:

1. The performance difference is small in practice. This is because most hard real-time systems also include a number of soft real-time tasks that must not obey their deadlines all the time.

2. Stability is easier to achieve with the RM algorithm. RM ensures the schedulability of the hard real-time tasks by simply assigning them a high priority. This guarantee is especially useful when transient errors or overload occur. In contrast, EDF changes the priority of each task dynamically, which makes the missing of deadlines more complicated to handle.

3. RM is straightforward to implement. A RM-based scheduler is more compact and faster, which reduces the overhead caused by the scheduling process.

One of the key research goals of the AMIDAR project is to develop a general-purpose Java processor that favors the fairness among threads over the real-time metrics. Therefore, the MLQ algorithm with the RR inter-queue scheduling strategy was chosen as the basis of the thread scheduler of the AMIDAR processor. However, if the priority-based inter-queue scheduling strategy was implemented, this scheduler could also support the RM algorithm directly. By default, the scheduler assigns a thread twice as much processor time as a thread with the next lower priority. In addition, it also supports resetting the priority of a thread at runtime (i.e. dequeuing and re-enqueuing the thread). This feature is essential for solving the *priority inversion* problem described below.

Priority Inversion

When combining a preemptive, priority-based scheduling algorithm with a synchronization mechanism, just like in Java, a thread can be preempted by a lower-priority thread indirectly, which is traditionally referred to as *priority inversion* [63]. The example below is intended to explain this issue in the context of the AMIDAR processor in more detail.

Assume that 3 threads T_0, T_1 and T_2 with priority p, $p+1$ and $p+2$ have been executed on the AMIDAR processor for several scheduling rounds. Due to the scheduling algorithm used by the AMIDAR processor, the 3 threads are executed in a fixed order in each round, namely $T_2 \rightarrow T_1 \rightarrow T_0$. The lengths of the time-slices of T_0, T_1 and T_2 are t, $2t$ and $4t$ respectively. Figure 2.3 shows a scenario in which the priority inversion problem arises between t_1 and t_2. During this time, T_2 is blocked by T_1, which effectively inverts their relative priorities. The entire process is described step-by-step in the following.

First, at t_0 in round R_n, T_0 acquires the monitor of some shared resource r. As T_2 tries to enter the same monitor at t_1 in round R_{n+1}, it blocks and is removed from the ready thread queue, which causes the scheduler to make a new schedule for T_0 and T_1. From R_{n+2} to R_{n+4}, both threads are executed in turn. At t_3, T_0 releases r and terminates at the end of R_{n+4}. Consequently, T_2 is attached to the ready thread queue again at t_3, which leads to a new schedule as shown in round R_{n+5}. Clearly, the reason for the priority inversion is that T_1 is preferred by the scheduler from R_{n+2} to R_{n+4} due to its higher priority.

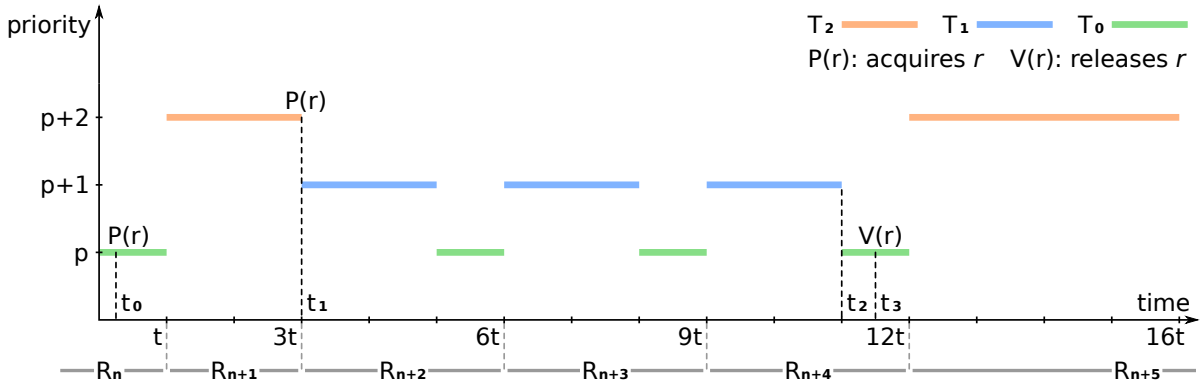


Figure 2.3: Priority inversion

In the example above, the priority inversion takes $6t$ execution time, which would be tolerable in most cases. However, if there were n further threads with priority $p+1$, the duration of the priority inversion would become $6t \cdot n$. Such a delay might cause serious problems in certain circumstances. To solve the priority inversion problem, two well-known approaches have been proposed, namely *priority ceiling* and *priority inheritance*. The common idea of these approaches is to raise the priority of the owner thread of a monitor temporarily, if a higher-priority thread is blocked by the monitor.

Priority Ceiling: The priority ceiling protocol [63] assigns each shared resource a ceiling, which is the highest priority of any thread that may need the resource. The highest ceiling of all resources that are currently being locked is said to be the *system ceiling*, where the corresponding resource and its owner thread are referred to as r_{sc} and T_{sc} below. A thread T may lock a resource only if its priority is strictly higher than the system ceiling regardless of whether the resource it needs is r_{sc}. Otherwise, T blocks and the priority of T_{sc} is raised to that of T, if T has a higher priority than T_{sc}. The priority of T_{sc} falls back to its original value as soon as r_{sc} is released. The priority ceiling emulation protocol is a simplified version of the original protocol [22]. In this protocol, once a thread locks a resource, its priority is raised to the ceiling of the resource immediately.

Priority Inheritance: The priority inheritance protocol [96] aims to solve the priority inversion problem without the need for knowing the ceilings of all shared resources previously. If a thread T is blocked by some resource that has been locked by a lower-priority thread, the priority of the owner thread is raised to that of T until the resource is released.

The major advantages of the priority ceiling protocol is that it avoids both priority inversion and deadlock. In contrast, the priority inheritance protocol cannot solve the deadlock problem. However, a deadlock should be considered as a design error and be fixed manually rather than by a runtime system. Furthermore, the priority ceiling protocol requires that the ceiling of each shared resource is already known at compile time and stays fixed at runtime, which is impossible without changes to the original Java thread and synchronization model. This is because the priority of a Java thread may be modified arbitrarily throughout its lifetime (e.g. caused by some random external event), which has the consequence that the ceiling of a resource needed by this thread cannot be determined by the Java compiler. Therefore, the thread scheduler of the AMIDAR processor implements the priority inheritance protocol to obey the standard Java specification.

Figure 2.4 illustrates how the priority inversion in the example above is avoided by exploiting the priority inheritance protocol. As shown, T_0 inherits the priority of T_2 at t_1. Due to this, T_0 is preferred by the scheduler in round R_{n+2} and is assigned a time-slice whose length is equal to $4t$. At t_2, it releases r, causing T_2 to be added into the ready thread queue again. At the same time, the priority of T_0 falls back to the original value. Note that the current version of the scheduler does not change the length of the time-slice of a thread after the thread has been started. This means that T_0 could run further until t_4. However, in this example, T_0 finishes its task at t_3 and therefore terminates. Through the use of the priority inheritance protocol, the blocking time of T_2 is reduced by $6t$.

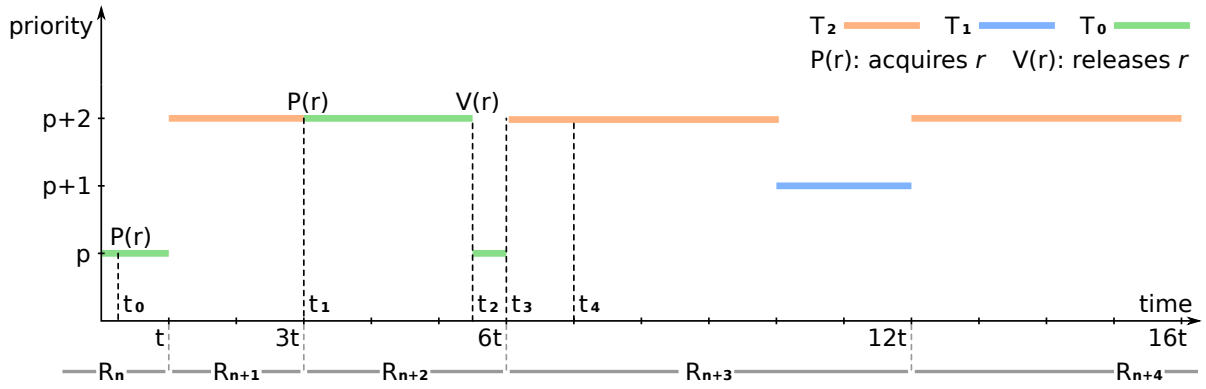


Figure 2.4: Priority inheritance

2.3.5 Lock Models

Thread synchronization is one of the main performance bottlenecks when running Java programs and consumes an average of 19% of execution time in some early version of JRE from SUN [44]. Even in a single-threaded program, a lot of time can be spent performing unnecessary synchronization. This is because Java is an explicitly multi-threaded language and its class libraries must be thread-safe so that they can be executed properly in a multi-threaded environment. As such, the methods of a class through which the shared data fields of the class are accessed must be declared as synchronized. Using thread-safe classes in a single-threaded program or locally within a thread may lead to a substantial performance degradation of up to factor of two [11]. Therefore, tremendous efforts have been devoted to solving this issue, including:

- Optimizing the underlying lock models used to implement the monitor construct to speed up synchronization [3, 11, 30, 36, 42, 55, 61, 78].
- Eliminating unnecessary synchronization operations at either compile time or runtime [5, 19, 60, 73, 85, 88].
- Providing thread-unsafe class libraries to allow for individual determination of whether synchronization is needed.

The latter two points above fall beyond the scope of this thesis and therefore are not described in detail. In the following, the discussion is focused on the major lock models and the possibility to implement them in hardware.

Fat Lock

The low-level implementation of the monitor construct needs to include a multi-word data structure holding all necessary information to support the complex monitor semantics, including queuing blocked threads, executing **`wait`**- and **`notify`**-methods etc. Such a data structure is traditionally referred to as a *fat lock* [11]. It contains at least an owner thread pointer, a recursive lock count as well as two queues holding blocked and waiting threads respectively. The structure of a fat lock can be varied depending on concrete requirements. For example, the blocked and waiting thread queues can be merged into a single one [3, 30], or further data fields like inherited priorities can be introduced.

According to the Java specification, every object is associated with a monitor, which gives an intuitive impression that every object should contain such a complex data structure additionally. However, except in the very early version of the *Kaffe Virtual Machine* [108], this naive implementation has never been used in practice due to one of the important characteristics of the Java synchronization behavior: only a small part of objects are actually synchronized in real-world programs [11]. Therefore, reserving multiple words in each object to hold the entire fat lock structure would cause unacceptable memory redundancy. A solution for this problem, which was used in the early versions of both SUN and IBM JVMs is to keep all fat locks completely outside of the synchronized objects and store them in a central monitor cache that is typically implemented in terms of a free list. One of the main disadvantages of this approach is the missing link between objects and their locks. As a result, the lock of a given object must be looked up in the monitor cache, which is quite inefficient. Furthermore, the monitor cache itself is a shared resource and therefore must be locked during lookups to avoid race conditions caused by multiple modifying threads. Also, the fat locks held in the monitor cache need to be cleaned up either by a dedicated thread periodically or by the garbage collector, increasing runtime overhead.

Thin Lock

To speed up acquiring and releasing uncontented locks, especially when running single-threaded programs, Bacon et al. proposed the *thin lock* model [11] based on a 24-bit lock field included in the object header. This lock field is solely made up of a *shape bit*, a 15-bit owner thread ID and an 8-bit recursive lock count. As long as a lock is not contended, it stays in the thin lock mode (or *flat* mode) and its shape bit remains zero. A thread can acquire a thin lock, if the lock's owner thread ID is equal to 0, which means that the lock is free, or is equal to the ID of the acquiring thread. Otherwise, the lock is already owned by another thread. In this case, the acquiring thread enters a spin-locking loop and busy-waits until it obtains the lock. Then, the lock is inflated as follows: its shape bit is changed to one and the remaining 23 bits are overwritten with the ID of a fat lock that is allocated on the fly. The fat lock's ID corresponds to an index into a lookup table that holds pointers to all fat locks. Exploiting this lookup table, the fat lock can be addressed efficiently. Once a thin lock is inflated, it stays in this mode for the rest of its lifetime.

Tasuki Lock

The *tasuki lock* model [78] aims to address the two primary weaknesses of the thin lock model, namely irreversible inflation and busy-waiting. The thin lock model is intended to increase the performance of synchronization in the absence of any true concurrency and therefore does not support

deflation of fat locks. However, the measurements of multi-threaded benchmarks show that most lock contentions are temporary in Java [78]. Therefore, in the tasuki lock model, if an inflated lock without any blocked or waiting thread is released, it is deflated back to a normal thin lock, by simply resetting its entire lock field to zero.

To eliminate busy-waiting caused by lock contention, the tasuki lock model adopts a single bit flag called *flat-lock bit* (FLC bit). The FLC bit is built into the object header as well, but outside the lock field. This is because the FLC bit should be set by a contending thread, while the lock field may solely be modified by the owner thread for the purpose of data consistency. As a thread fails to obtain the thin lock of an object, it sets the FLC bit of the object, creates a new fat lock, attaches itself to the blocked thread queue of the fat lock and inserts the fat lock into the corresponding lookup table. Then, this thread suspends itself and starts waiting. Upon releasing the lock, the owner thread checks the FLC bit. If it is set, the owner thread traverses the lookup table to find the corresponding fat lock and then signals all blocked threads. Once a resumed thread obtains the lock, it inflates the lock just like in the original thin lock model.

Sable VM Lock

The *Sable Virtual Machine* (Sable VM) [109] uses a slight variation of the tasuki lock model that moves the FLC bit from the object header into the internal data structure of a thread [36]. The reason for this modification is that the space in the object header is very expensive and even a single bit could cause notable memory overhead if a large number of small objects are generated. Since the FLC bit can be modified by multiple threads, a dedicated mutex is employed to ensure mutually exclusive access to it. Also, a thread may own the locks of different objects at the same time. Therefore, the thread structure must additionally contain a linked list holding tuples, each of which combines one of its lock objects with a thread blocked by the lock of the object. As a thread releases the lock of an object O, it first checks the FLC bit. If the bit is set, it goes through its tuple linked list and signals all blocked threads regardless of whether they are blocked by the lock of O. In the meantime, the locks of all objects held in the list are inflated, except that of O, because it will be inflated by its new owner thread.

Biased Lock

Biased lock is also known as *reservation lock* [55] or *lazy lock* [42]. It aims to optimize the thin lock model by exploiting the *thread locality* of Java locks.

> Thread Locality [55]: *For a given lock, if its locking sequence contains a very long repetition of a specific thread, the lock is said to exhibit thread locality, while the specific thread is said to be the dominant locker.*

The key idea of the biased lock model is to reserve a lock for its dominant locker thread so that this thread can access the lock more efficiently. Based on the observation that more than 75% lock operations in multi-threaded programs are performed by the first owner threads in the first repetitions [55], the biased lock model considers the first owner thread of a lock as its dominant locker. As an object is locked by some thread for the first time after its creation, the ID of the thread is written into the lock field of the object. This thread ID is kept in the object's lock field as long as no contention for

the lock occurs, regardless of whether the object is actually locked or not. During this time, the lock is said to be in the reverse mode. Once this mode is canceled, the lock may not go back to it again. In the reserve mode, the dominant locker only needs to increment or decrement the recursive lock count when acquiring or releasing the lock.

CACAO VM Lock

Krall et al. [61] proposed a lock model implemented in the *CACAO Virtaul Machine* (CACAO VM) [107]. Rather than a monitor cache, this model utilizes a small hash table to keep all fat locks. The physical address of an object is used to calculate the hash table index. This eliminates the memory overhead caused by the lock field included in the object header at the expense of calculating the hash table index on every lock access and handling overflows in the hash table. Once a fat lock is inserted into the hash table, it will not be removed and may be reused for another object after the current object is not synchronized anymore. This greatly reduces the total number of fat locks generated at runtime.

JOP Lock

The *Java optimized processor* (JOP) uses a quite unique lock model [104]. A single global lock is shared by all threads, which only consists of a recursive lock count field. Once a thread acquires the lock of an arbitrary object, the context switching function of the processor is disabled and the lock count of the global lock is set to one. This means that the thread owns not just the lock but the entire processor and no other thread can preempt its execution. In the meantime, the thread may acquire the lock of any further object, leading to the incrementation of the lock count of the global lock. Each time a lock is released, the lock count of the global lock is decremented. Once it reaches zero, the context switching function of the processor is enabled again. The advantages of this model are deadlock freedom and ease of implementation. However, it does have several limitations, including the lack of support for the **`wait`**- and **`notify`**-methods as well as the blocking of all other threads that even have no need for any lock or have higher priorities.

AMIDAR Lock

The basic idea of the AMIDAR lock model is similar to that of the CACAO VM: holding all active fat locks together in a small BRAM-based monitor table and mapping an object to its lock with the help of a *content-addressable memory* (CAM) which serves as a lookup table. A fat lock is considered to be active, if it is owned by some thread or has at least a blocked or waiting thread. Once a fat lock becomes inactive, its slot in the monitor table is freed and may be reused by another fat lock.

The major concern of the AMIDAR lock model is the peak number of active locks, because the size of the monitor table is fixed throughout the entire runtime and an overflow causes an exception to be thrown. Although an FPGA allows for customization on an application-by-application basis, the hardware usage could be too high to be acceptable if the monitor table was too large. Furthermore, increasing the size of the monitor table reduces the maximum clock frequency of the CAM dramatically, which could limit the performance of the whole system. Fortunately, according to the measurements of real-world programs, even a highly multi-threaded HTTP server written in Java [52] uses only up to 25 active locks simultaneously, as it serves parallel requests from 7 hosts, amounting to about one megabyte

split across 200 files for each request [61]. To gain a deeper insight into this issue, the lock usages of other 4 well-known multi-threaded benchmarks were profiled by using an instrumented version of CACAO VM 1.6.1. Table 2.1 summarizes the profiling results.

Benchmark	SPEC	DaCapo		
	jbb2005	xalan	hsqldb	lusearch
Workload	8 warehouses	large	large	large
Active locks	33	35	12	38

Table 2.1: Peak number of active locks

SPEC jbb2005 [98] models a wholesale company with 8 warehouses, each of which needs to handle both operations initiated by customers and company internal affairs. *Xalan* of the benchmark suite *DaCapo* [15, 28] simulates a typical server XSLT load, which performs XML to (X)HTML transforms as part of a presentation layer. A number of threads are employed to respond to parallel queries from different hosts. *DaCapo hsqldb* runs a SQL database engine to perform transactions against a model of a banking application. *DaCapo lusearch* looks up given keywords over a corpus of data, using multiple threads, each of which searches a large index for about 3500 distinct words.

2.4 Priority Queue Architectures

The centerpiece of a hardware scheduler is the *priority queue* (PQ) that holds multiple threads and sorts them in descending order of priority. This means that a PQ is always capable of yielding a thread with the highest priority on request without delay. The most common PQ architectures include *binary tree of comparators* (BTC) [79, 87], *shift register* (SR) [26, 111], *systolic array* (SA) [64, 65] and FIFO [21, 25]. In the following, each of these architectures is described, where N and P are used to represent the PQ size and the number of different priority levels.

Binary Tree of Comparators Priority Queue

Figure 2.5 shows the basic structure of a BTC-PQ that contains a queue memory with N thread buffer blocks, a comparator tree of depth $log_2 N$ and a control logic with a feedback connection from the last node of the comparator tree. A thread buffer block includes two registers that hold thread ID and priority respectively. Also, the block has a flag bit that indicates whether it is idle. The ID and priority of a new thread are broadcast to all of the N blocks through a common bus. According to the flag bit of each block, the control logic (decoder) selects a free block for saving the thread being enqueued and sets the write-enable signal of the selected block. Each node in the comparator tree picks the thread with the higher priority from both input ports and forwards it to the next node. In this way, the last node outputs the highest-priority thread. Using the feedback connection, the control logic will reset the idle flag of the buffer block holding the selected thread.

The advantage of this architecture is that the comparator tree can be shared by multiple queue memories (e.g. by the ready thread queue and the blocked thread queue). After changing the queue memory, the comparator tree can always provide a scheduling result in constant time. Another advantage of the BTC-PQ architecture is that the priority of a thread can be easily changed at runtime by broadcasting its

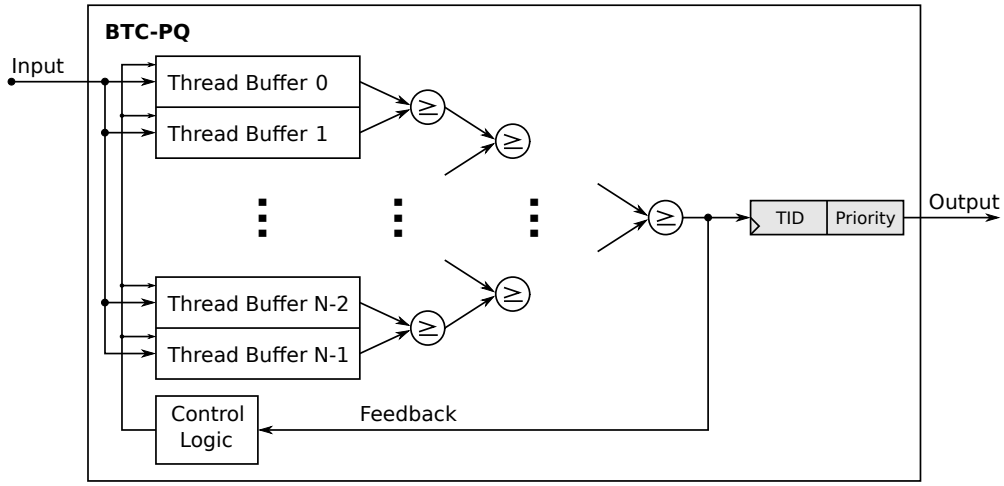


Figure 2.5: Binary tree of comparators priority queue

ID and the new priority value to all thread buffer blocks. Only the block with the matched ID updates its priority register. The scalability of this architecture is limited by N because of the depth of the comparator tree and the bus loading problem caused by distributing the input value to every thread buffer block. Another problem of the BTC-PQ architecture is that it considers only the priorities of all in-queue threads during scheduling and does not perform any temporal ordering like RR or FIFO on the threads. Therefore, it is not suitable for a general-purpose thread scheduler due to the lack of fairness.

Shift Register Priority Queue

As shown in Figure 2.6, a SR-PQ is a linked list that contains a fixed number of shift register blocks holding the information about in-queue threads. The key feature of the SR-PQ architecture is the self-ordering of threads based on their priorities. This is achieved through a local sorting control distributed into each shift register block.

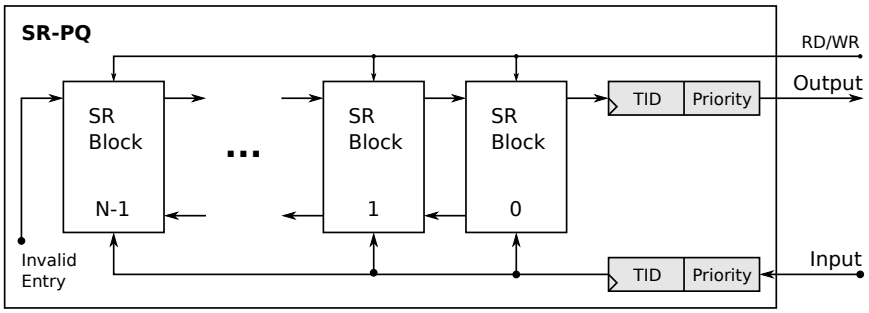


Figure 2.6: Shift register priority queue

Figure 2.7 illustrates the internal structure of a shift block that is made up of a multiplexer, a comparator, a control logic and a register that stores the information (ID and priority) about a thread. The thread register is connected directly to both neighbor blocks and the output of the comparator is forwarded to the left neighbor only. The 5 inputs are the access control (READ/WRITE), the comparison result of the right block, the new thread entry and the thread information from the left and right neighbor blocks.

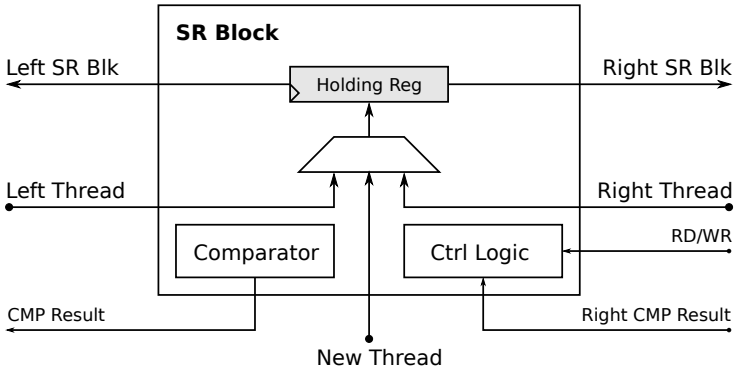


Figure 2.7: Shift register block

As a new thread is enqueued, the WRITE command along with the ID and priority of the thread are broadcast to all blocks. Each block compares the given priority with the one held in its thread register. The output of the comparator is set to TRUE, if the new thread has a higher priority than the current one. According the the comparison result, three possible decisions can be made:

- The block does nothing, if the new thread has a lower priority than the current one, i.e. the local comparison result is FALSE.
- The block replaces the current thread with the new one, if the local comparison result is TRUE and the comparison result of the right block is FALSE.
- The block replaces the current thread with the thread of the right neighbor block, if both comparison results are TRUE.

As a result, only one block keeps the new thread. All blocks on its right side stay still and the values of other blocks are shifted to the left. In this way, the entire queue is automatically reordered on each enqueue operation and the rightmost block (PQ-head) contains always a thread with the highest priority. Also, the FIFO order of all threads with the same priority is maintained, which is another advantage of the SR-PQ architecture. During a dequeue operation, the value held in PQ-head is simply read out and the values in the rest of the blocks are shifted one block to the right. Both of the enqueue and dequeue operations need a single clock cycle only.

One of the major disadvantages of SR-PQ is the bus loading problem which the BTC-PQ architecture also has. Broadcasting the new thread entry to all blocks in a single clock cycle requires extra intermediate buffers (e.g. LUTs in FPGAs) to be added to the routing path, which limits the maximum clock frequency. Another disadvantage is that SR-PQ, unlike BTC-PQ, cannot be shared between different thread queues. Consequently, each thread queue needs a dedicated SR-PQ instance of size N, which would cause unacceptable resource usage in certain circumstances. For example, each monitor in Java is associated with two queues holding the threads waiting for it and threads blocked by it respectively and a Java application can use a number of monitors at the same time. This means that there should be twice as many SR-PQ instances as monitors, which is not realistic in practice. The third disadvantage of SR-PQ is that it only supports the enqueue and dequeue operations and does not allow for random access to an arbitrary block in it, which is essential for changing the priority of a thread. For example, if the currently running thread changes the priority of another thread that is saved somewhere in middle

of the ready thread queue, the target thread should be removed from the queue, assigned a new priority and re-enqueued, which cannot be realized due to the lack of the support for random access.

Systolic Array Priority Queue

Figure 2.8 demonstrates the structure of a SA-PQ, which is quite similar to that of a SR-PQ and also contains a linked list of identical blocks. The key difference between them is the way how a thread is enqueued. Unlike a SR-PQ, a SA-PQ does not broadcast the new thread and WRITE command to all blocks at the same time, but passes them to the rightmost block only. If the new thread has a higher priority than the existing one, the new thread is written into the thread register of the block and the old one is passed on to the right neighbor block. Otherwise, the new thread is forwarded. Regardless of which thread is output by the block, the WRITE command is simply sent to the next block. This compare-and-send process is performed by one block per clock cycle and repeated until the end of the queue (PQ-tail) is reached. This indicates that enqueuing a new thread takes multiple clock cycles in stead of one. However, the PQ-head contains always a thread with the highest priority. Also, the FIFO order of all threads at the same priority level is maintained automatically. Thus, from the output point of view, a SA-PQ completes the enqueue operation just in a single clock cycle.

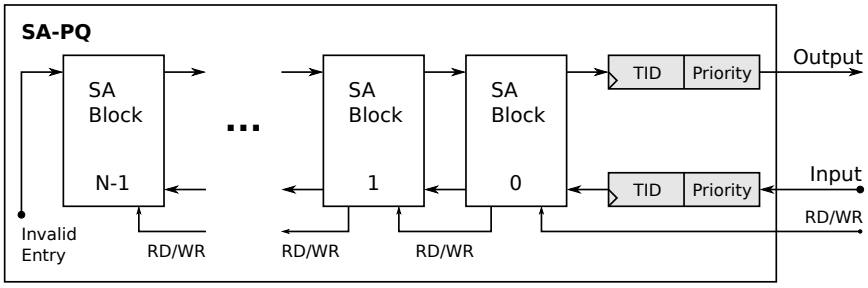


Figure 2.8: Systolic array priority queue

Unlike a shift register block, a systolic array block has double thread registers and multiplexers as shown in Figure 2.9, which are used to swap the input thread and the current thread according to the result of the comparator. The holding register keeps the thread that should stay in the block, while the temporary register holds the thread that needs to be sent further. The main advantage of the SA-PQ architecture is that it eliminates the bus loading problem. However, this architecture uses twice as much storage as the SR-PQ architecture to save the same number of threads. Also, it has the sharing and random access problems just like the SR-PQ architecture.

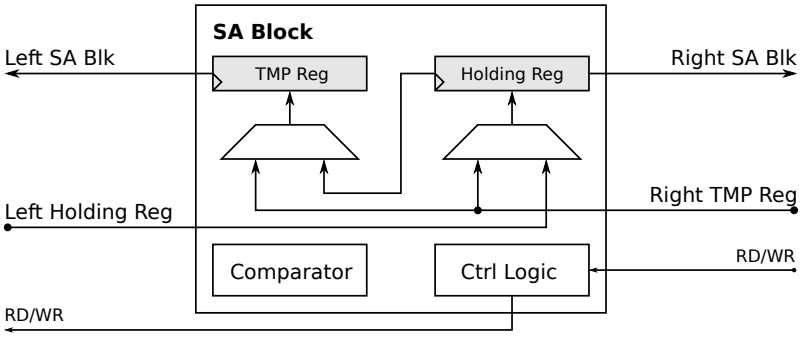


Figure 2.9: Systolic array block

FIFO Priority Queue

The FIFO-PQ architecture is the straightforward implementation of the MLQ algorithm. It contains a dedicated FIFO-queue for each priority level. Also, it has a queue selector on both input and output sides. The input queue selector needs to pick a queue according to the priority of a new thread to perform the enqueue operation. The output queue selector can be customized to meet different requirements. The simplest implementation is that it outputs only the threads held in the highest-priority FIFO-queue that is nonempty, which could cause the starvation of threads with lower priorities. Another possible solution is to select all FIFO-queues in a round-robin manner.

One of the problems of the FIFO-PQ architecture is the large resource usage, since each priority level needs a full-sized FIFO-queue. Thus, the scalability of a FIFO-PQ is restricted by both N and P. Another drawback is that a single FIFO-PQ instance cannot be shared by multiple thread queues. Due to these two issues, it is impossible to use this architecture to implement the waiting and blocked thread queues of the monitor construct of Java. Also, the classical implementation of a FIFO-queue does not support random access to an arbitrary entry in the queue. As a result, changing the priority of a thread is difficult to realize, because this requires the ability to remove the thread from any position of any FIFO-queue and add it into another FIFO-queue.

J. Agron et al. [4] proposed a variation of the FIFO-PQ architecture that holds the IDs of both head and tail threads of each FIFO-queue only. The data of all threads are saved in a central thread table separately. The ID of a thread corresponds to its thread table index. The thread entries in the table which have the same priority are linked together in the FIFO order to form a list. This greatly reduces the resource usage of a FIFO-PQ, but at the expense of complications for the enqueue and dequeue operations. Also, using this FIFO-PQ architecture, it is relatively simple to change the priority of a thread, because the thread can be directly located in the thread table using its ID.

3 Related Work

3.1 Java Processors

Since the mid-1990s, a number of Java processors have been developed for either commercial or academic purposes. A detailed survey on these processors can be found in [93]. Although they all target embedded systems with limited recourses, each of them provides some quite unique characteristics. In this section, several representative Java processors are selected and presented from different perspectives, including *picoJava-II* [80], *Java multithreaded processor* (Jamuth) [113], *Java optimized processor* (JOP) [92] and *secure hardware agent platform* (SHAP) [83]. Table 3.1 provides a brief comparison of them and the AMIDAR processor.

	Pipeline depth	Object cache	Garbage collector	Thread scheduler	Stack realization	Java standard
picoJava-II	6	No	SW	SW	Stack cache	J2SE 1.2
Jamuth	5	No	SW	HW/SW	Stack cache	J2ME CLDC
JOP	4	Yes	SW [1]	SW	Stack cache	J2ME CLDC
SHAP	4	Yes	HW	SW	On-chip stack	J2ME CLDC
AMIDAR	0	Yes	HW	HW	On-chip stack	J2SE 1.4 [2]

[1] A hardware garbage collector was designed by Gruian et al as an expansion module for JOP [41].
[2] The AMIDAR processor currently does not support dynamic class loading and linking.

Table 3.1: Overview of selected Java processors

All of the selected processors are based on a pipelined RISC-like microarchitecture constructed around a stack unit. This allows a straightforward translation of Java bytecode to their microcode (i.e. the native instruction sets of these processors). In such a processor, one bytecode is typically mapped either to a single microinstruction or to a sequence of microinstructions called microprogram. Also, it is common for these processors to interpret complex bytecodes such as **`new`** or **`monitorenter`**, using software traps. Furthermore, Jamuth, JOP and SHAP do not include a floating-point unit. Therefore, they have to realize floating-point arithmetic in software as well.

picoJava-II

Among all processors shown in Table 3.1, picoJava-II of Sun Microsystems is the only one designed for ASICs. It is the succeeding version of the picoJava processor [77] released in 1997. Unlike the other three processors described below, picoJava-II aims to provide a general solution for a broad spectrum of various embedded systems. Therefore, it fully supports J2SE and allows any class to be loaded and linked at runtime.

The stack cache of picoJava-II is implemented by using a register file with 64 entries. The stack management unit automatically spills the cache upon an overflow and fills it upon an underflow. To reduce the latency incurred by memory accesses, a 2-way set-associative data cache is introduced between the stack cache and the main memory, which has the default size of 16 kilobytes. From this point of view, the data cache can also be considered as a second level stack cache.

Besides the data on the stack, the data cache also buffers object fields and array elements. picoJava-II supports both direct and indirect object addressing schemes described in Section 2.3.2, whereas the data cache is physically addressed only. Thus, a delay of at least two extra clock cycles will be caused by every object access when using handles [81].

Although complex management tasks like garbage collection and thread scheduling need to be realized in software, picoJava-II provides several low-level mechanisms to assist with performing such tasks. For example, it triggers a writer-barrier trap under certain conditions when a reference field of some object is overwritten. In the trap handler, corresponding actions can be taken depending on the garbage collection algorithm used.

Additionally, picoJava-II contains two breakpoint registers which are employed to provide debugging assistance at the hardware level directly. To add an instruction breakpoint or a data breakpoint, an instruction memory address or a data memory address needs to be written into either of the registers. In the debugging mode, picoJava-II performs breakpoint check on every instruction fetch or data access and triggers the predefined breakpoint trap upon an address match.

Jamuth

Unlike picoJava-II that is intended as a general-purpose processor, Jamuth is aimed in particular at hard real-time systems. The key feature of Jamuth is that it consists of four hardware thread slots each of which has dedicated fetch and decode stages and shares the rest of the pipeline with the other three slots. Therefore, Jamuth can execute four threads simultaneously by fetching one instruction from each of them in a single clock cycle. To determine which of these instructions needs to be passed on to the execute stage, a hardware thread scheduler is integrated into the pipeline directly. In Section 3.4 below, this scheduler is described in detail.

Among the four hardware thread slots, one is reserved for the garbage collector and two are free for special real-time threads. The remaining slot is adopted to execute regular threads without real-time requirements. A simple software thread scheduler is responsible for scheduling these non real-time threads. The stack cache of Jamuth is made up of two BRAMs and includes a total of 2048 entries. Unlike picoJava-II, Jamuth performs spills and fills of the stack cache with microprograms. Every hardware thread slot is assigned a portion of the stack cache. At runtime, any thread running within a hardware thread slot may only access the stack cache portion associated with this slot.

Another important difference between Jamuth and the other processors shown in Table 3.1 is that it does not support indirect object addressing. Thus, the garbage collector of Jamuth does not compact the heap in order to avoid the significant overhead caused by updating references to reallocated objects. Also, the lack of handles causes that no logically addressed object cache can be built into Jamuth.

JOP

Similar to Jamuth, JOP was also designed for embedded real-time systems that must ensure time-predictable execution of programs. Its major focus from the beginning has been to enable and simplify accurate *worst case execution time* (WCET) analysis. To achieve this goal, several key design decisions were made: 1. only a fully-associative object cache may be used, which is explained in the following

section. 2. threads may only be defined statically, i.e. dynamic creation of thread is not allowed. 3. there is only a single global lock in the entire system, as described in Section 2.3.5.

JOP has a SRAM-based stack cache of size 128 entries. To increase the access performance, the two topmost values of the operand stack are buffered in two separate registers rather than in the cache. A simple control circuit updates both of the registers automatically. Like in Jamuth, spills and fills of the stack cache are also carried out by using microprograms.

SHAP

SHAP is an enhanced version of JOP [94] and aims to achieve a higher performance than the original design through several architectural modifications. First, it contains a dedicated heap management module that performs object allocation and garbage collection autonomously. The centerpiece of this module is a general-purpose microprocessor that runs a C-programed garbage collector. More details about the heap management of SHAP are given in Section 3.3. Second, the whole Java stack of SHAP resides in the on-chip memory instead of the external main memory. Therefore, the stack cache built into JOP is replaced with a stack module consisting of multiple BRAMs. By default, there are a total of 2048 stack entries available for program execution. Furthermore, SHAP supports dynamic creation of threads and allows the use of multiple locks at the same time.

3.2 Object Caches

Based on the logical addressing scheme described in Section 2.3.2, an object-based memory system [115, 116, 120, 123] provides architectural support for objects directly. One of its major components is a logically addressed object cache that allows fast access to objects upon cache hits. The key design goal of such a cache is to increase the average hit rate. Besides several general factors like the cache line size or the associativity (i.e. the way number), there is an additional factor that can affect the hit rate of a logically addressed cache significantly, namely the way of generating the cache index. As discussed in Section 2.3.2, when a field of an object is accessed, a logically addressed cache should use a subset of the object's handle bits and a subset of the field's offset bits at the same time to create the cache index. Which portion from each of the handle and the offset is selected is the key design decision that needs to be made. The selected offset bits should maximally spread out the cache lines assigned to a single object, reducing intra-object conflicts, while the selected handle bits should distribute cache lines occupied by different objects across the cache, reducing inter-object conflicts. For ease of representation, a logically addressed object cache is simply referred to as an object cache in the following discussion.

In an object-based memory system, both object handle and field offset are typically represented in terms of a 32-bit unsigned integer. Since the handle of an object is the object's unique identifier, it must be taken by an object cache completely to tag cache lines. In contrast, previous research shows that the average object size is only about 48 bytes [127], which means that using the lowest 8 offset bits will cover the majority of non-array objects [120]. Under consideration of arrays, more offset bits are needed, but up to 12 offset bits will be enough in most cases [116]. For this reason, most existing object caches only use several lowest significant offset bits and simply discard the remaining bits. As a result, an extra mechanism is necessary to handle larger objects. One possible solution is to break a large object into smaller objects at compile time [120, 123]. Another solution proposed in [115] is to perform an

additional check on each cache access. Once a larger offset is detected, the cache simply redirects this access to the main memory.

In the following, four index generation schemes are discussed based on a sample object cache introduced in [123], which has a 32-bit handle and a 10-bit offset as input[1]. Figure 3.1 illustrates these schemes, each of which is explained briefly below:

1. Concatenation of the most significant bits (MSBs) of the handle with the MSBs of the offset.
2. Concatenation of the least significant bits (LSBs) of the handle with the MSBs of the offset.
3. Concatenation of the LSBs of the handle with bits in the middle of the offset.
4. Performing a logical XOR on the LSBs of the handle and the MSBs of the offset.

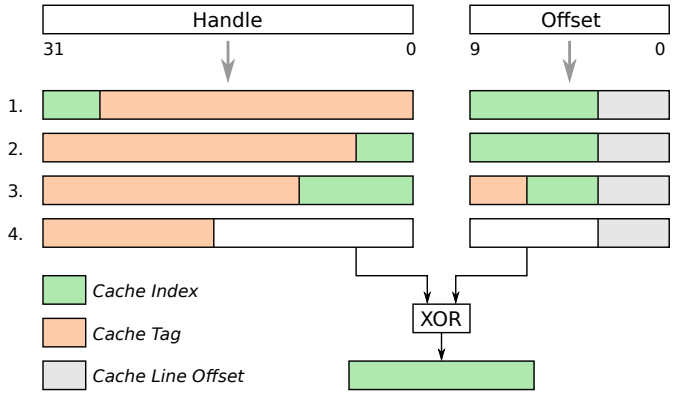


Figure 3.1: Object cache index generation schemes

These schemes were explored using a simulator developed for the *MUSHROOM* system [120]. According to the simulation results, the last scheme provides the best average hit rate. Therefore, it was suggested to be used either for implementing the memory system of a Java processor [115, 116], or for improving the functionality and efficiency of heap management of a JVM [123]. The first scheme causes notable cache collisions since most object handles map to the same set. The second and third schemes are somehow varieties of the same one. The major disadvantage of both is that the hit rate can be heavily affected by the object size. Using the second scheme, small objects map to the same set. The third scheme causes that fields of a large object map to the same set. We assumed that this disadvantage could be eliminated if the object cache could adjust the portion taken from the offset dynamically, according to the object size. Upon this assumption, a cache index generator using the concatenation operation was developed, which is described in Section 4.5.3 in more detail.

Several real-time Java processors like JOP [46] and SHAP [129] have to use a fully-associative cache that provides a suboptimal hit rate, because cache index generation based on a hash function like XOR would defeat the WCET analysis [45]. This cache simply uses object handles to tag cache lines and field offsets to address words inside cache lines. The primary problem with this cache is that each object may only be cached in a single cache line and thus the fields at higher offsets cannot be cached, if an object is larger than the cache line size. This implies that this cache is suitable for small objects only.

[1] Note that although the original offset consists of 32 bits, the highest 22 bits are ignored by the cache.

3.3 Hardware Garbage Collectors

The first attempt to perform hardware-assisted garbage collection was found in early 1990s, when Nilsen and Schmidt proposed a *garbage-collected memory module* (GCMM) [91] for C++, which aims to support efficient object space management. The key components of the GCMM include a general-purpose microprocessor and two *object space managers* (OSMs), which communicate with each other over an internal bus. The microprocessor is employed to facilitate object allocation and to perform garbage collection based on Baker's incremental copying algorithm discussed in Section 2.3.3. The OSMs treat objects as first-class entities in the main memory and allow them to be created and accessed in one memory cycle [76]. To support precise garbage collection, the GNU C++ compiler was extended so that it can generate a bitmap for every class used in a program. This bitmap indicates which fields defined in the class hold pointers and is sent to the GCMM upon allocating a new object of the class. Based on the information provided by the bitmap, an extra flag bit is associated with each field of the newly created object.

The GCMM partitions the main memory into two semi-spaces to realize the chosen garbage collection algorithm. Either of these spaces is managed by an individual OSM. New objects are allocated from the active space by using a bump-pointer. Once there is not enough contiguous memory in the active space to satisfy an allocation request, both spaces are flipped immediately so that the mutator can proceed without being suspended. Then, live objects are copied from the inactive space and placed side by side at the beginning of the active space, while new objects are allocated successively at the end of the active space. This implies that the bump-pointer will move towards the beginning of the active space during the copying process. If the mutator tries to access an object that has not been copied yet, the read-barrier will first redirect this access to the inactive space and then copy the object into the active space. Each time after an object O has been copied, its header is overwritten with *forwarding address* to its copy in the active space. If any pointer to O is detected while copying another object, it is replaced with a pointer to the copy of O according to the forwarding address held in O's header. In this way, references to live objects are updated incrementally.

The *active memory processor* (AMP) [100] is intended to increase performance and predictability of dynamic memory management in real-time Java embedded systems. It consists of an object allocator proposed in [24] and a reference counting garbage collector. The allocator utilizes a bitmap and two *complete binary trees* (CBTs) to perform speedy object allocation. Each bit contained in the bitmap corresponds to a fixed-length block of contiguous memory. Such a memory block is the smallest data unit allowed in an AMP-based memory management system. The assertion of a bit indicates that the corresponding memory block is currently occupied by some object. The CBTs of the allocator are pure combinational circuits. Based on the information held in the bitmap and the size of an object that needs to be allocated, they can locate the position of the first chunk of successive free memory blocks into which the object will fit in constant time. Then, with the help of a *bit-flipper* that is also a combinational circuit, the bits of the found memory blocks are flipped from zero to one. Later, upon deallocating the object, these bits are simply flipped again, which can be performed in constant time as well.

The garbage collector of the AMP maintains a 3-bit reference count for every object, which greatly reduces the overhead caused by reference counting operations. This design decision was made regarding the observation that most objects in embedded applications have a maximum reference number of 6. Additionally, the garbage collector employs a bitmap to store the size information about each object.

All bits contained in this bitmap are initialized with one. As an object is allocated, only the bit of the last memory block assigned to it is set to zero, i.e. the object's boundary is recorded. Consequently, the garbage collector can easily determine the size of a given object according to its base address and its boundary recorded in the bitmap. Once the reference count of an object becomes zero, the garbage collector deallocates it with the bit-flipper, which is much more efficient than a classical sweeping approach using a free list.

In practice, the AMP has three critical limits: 1. it is impossible to manage the entire memory with a single AMP because the bitmaps would otherwise be too large to be held in dedicated hardware and be handled by combinational circuits. 2. it cannot deal with reference count overflow. 3. it cannot recognize reference cycles just like any other reference counting garbage collector. To overcome the first limit, the main memory needs to be partitioned into multiple small segments with individual bitmaps. All these bitmaps are kept in a software lookaside buffer. Only those of the memory segment which needs to be accessed or has enough space to satisfy an allocation request are loaded into the AMP. A notable overhead would be caused if the AMP was reloaded frequently. To overcome the latter two limits, the AMP requires the assistance from a software mark-sweep garbage collector. Once the reference count of an object reaches 7, it will not be updated anymore, i.e. the object will not be deallocated by the AMP. If no memory segment has sufficient free memory to hold a new object, the software mark-sweep garbage collector will be triggered and clean up the main memory thoroughly in a stop-the-world manner. Furthermore, since the AMP targets CLDC-based devices only, it supports neither finalization of objects nor different reachability levels.

The *garbage collector unit* (GCU) proposed in [41] is a coprocessor developed for JOP, which is composed of a *background* module and a *reactive* module. To integrate the GCU into a JOP-based system, the background and reactive modules need to be connected to the main memory and the JOP core respectively. Inside the GCU, both modules communicate directly with each other through hand-shaking. The background module realizes the key functions of the GCU, involving object allocation and garbage collection based on an incremental mark-compact algorithm. The primary tasks of the reactive module include: 1. redirecting allocation requests from the JOP core to the background module. 2. assisting with the root set initialization at the beginning of the mark phase. 3. avoiding object access conflicts between the JOP core (the mutator) and the background module (the collector) in the compact phase. Since JOP is intended to be used with a predefined subset of the CLDC API only, the GCU also does not support object finalization and reference objects, like the AMP above.

In order to employ the GCU, multiple changes have to be implemented on the side of JOP. First, the application image generator must be extended to generate additional information about the distribution of reference fields of each class to support precise garbage collection. Next, the microprogram of every bytecode accessing objects like **`getfield`** must be modified to include extra operations that synchronize accesses from both of the mutator and the collector. Last, objects must be allocated through the GCU rather than from the main memory directly. Whenever a new object is created by the running program, an allocation request needs to be sent to the GCU via the dedicated interface between the JOP core and the reactive module. This causes that the background module allocates the desired object from the free memory area in a *bump-the-pointer* manner and then returns the updated bump-pointer back to the JOP core. If the remaining free memory falls below a predefined threshold, the JOP core triggers

a new garbage collection cycle explicitly. During the whole garbage collection process, new objects are simply allocated from the rest of the free memory area. However, after the occupied memory area has been compacted, all these objects, whether live or not, must be reallocated so that the occupied and free memory areas can be separated clearly. Finally, the bump-pointer is updated with the base address of the new free memory area.

In the mark phase of the garbage collection process, the background module traces references in the same way as Steele's concurrent mark-sweep collector described in Section 2.3.3. The stack required for this purpose resides in the main memory, which is initialized by the JOP core with the help of the reactive module. After all live objects have been marked, they are reallocated in the following compact phase. Every time an object moves, its handle table entry is updated using its new physical address. To avoid access conflicts between the collector and the mutator in this phase, a handle-based synchronization mechanism is introduced. Before the mutator performs a read or write operation on an object, it must first lock this object by sending the object's handle to the reactive module. If the object is not being overwritten or copied by the collector, the reactive module yields an acknowledgement so that the mutator can accomplish the required operation over the original connection between the JOP core and the main memory. As long as the object is locked, the collector may not reallocate or overwrite it and must stall until it is unlocked.

Like the GCMM, the *memory management unit* (MMU) of SHAP [82] is also constructed around a general-purpose microprocessor called ZPU [130]. This compact RISC core is employed to run a C-programed mark-copy garbage collector with the assistance of several special-purpose hardware components performing time-critical operations such as marking objects or scanning references. To enable precise garbage collection, SHAP adopts the bidirectional object layout used in the Sable VM, where all reference fields of an object are stored at negative offsets.

The MMU of SHAP partitions the main memory into multiple segments and maintains usage statistics for each of them, including the number of allocated objects and the size of occupied memory. At any time, only one of the segments can be used for the purpose of object allocation. Once this segment fills up, it is replaced with an empty one. The garbage collector is started periodically to mark live objects and to update the usage statistics of occupied segments. If the usage of some occupied segment falls below a given threshold, this segment is said to be sparse. In the copy phase, all live objects contained in sparse segments are moved into an empty segment. During this process, the handle table entries of these objects are updated with their new physical addresses on the fly.

To detect live objects, the garbage collector of SHAP implements a variant of Yuasa's sequential garbage collection algorithm introduced in Section 2.3.3. The major change is that the tri-color marking abstraction is realized by using the combination of a mark table and a mark FIFO instead of a mark stack. The mark table is initialized by the SHAP core with the references included in the root set. After that, the garbage collector traverses the table and checks every referenced object sequentially. If an object has not been encountered so far, it is marked and its reference is entered into the mark FIFO, otherwise the object is simply skipped. A dedicated hardware component removes every reference held in the mark FIFO and scans the corresponding object. If the object contains references to unmarked objects, these references are entered into the mark table and the mark FIFO at the same time. Whenever a FIFO overflow occurs, a status flag is set, which indicates that some live object has not been marked

and scanned yet. After the mark FIFO has become empty, the overflow flag is checked in addition. If it is asserted, the garbage collector has to traverse the mark table again, otherwise the mark phase completes. Note that this implementation of Yuasa's algorithm actually complicates the tracing process unnecessarily. Using a mark stack of the same size as the mark table would guarantee that all live objects can be marked in a single tracing pass.

Due to its software-based implementation and the associated flexibility, the garbage collector of SHAP supports soft and weak references. Depending on the memory usage observed at the beginning a garbage collection cycle, it either treats soft references as weak ones or simply ignores them. Thus, the following description focuses on the realization of weak references only. As a weak reference object is created, the highest two bits of the reference to its referent are used to hold the reachability level (01_2 represents weakly reachable and 11_2 softly reachable). Upon invoking the **`get`**-method on the reference object, these two additional bits are removed before the reference to the referent is returned. This means that a weak reference solely exists inside its reference object. When the garbage collector encounters a weak reference in the mark phase, it will abandon the referent directly without marking or scanning it. If the referent is only reachable through this weak reference, it will remain unmarked and therefore become eligible for garbage collection. In addition, all weak reference objects are collected into a linked list. At the end of the mark phase, the garbage collector traverses the list and removes every weak reference object whose referent is marked as reachable. This is because such a referent object can also be reached through another path of strong references. Accordingly, the rest of the weak reference objects are all cleared and optionally enqueued.

3.4 Hardware Schedulers

In Section 2.4, four *priority queue* (PQ) architectures are described, based on which several hardware schedulers have been implemented in prior work. This section gives a brief overview of these schedulers. In the following, N is used to represent the PQ size.

Schedulers based on BTC-PQ

The *real-time task manager* (RTM) proposed in [59] exploits the shareability of the BTC-PQ architecture to schedule ready and blocked threads, using a single BTC-PQ instance. Besides thread ID and priority, several other flags and attributes are added into each of the thread buffer blocks, including BLOCKED-flag and sleep time. To enable sharing the BTC-PQ, the structure of the comparator is slightly extended by introducing an enable-signal for both input ports. Only if both of the input threads are enabled, a comparison of their priorities is performed. The higher-priority thread is passed on along with a valid-signal to the comparator of the next level, where the valid-signal is connected to the enable-signal of the corresponding input port. If only one of both input enable-signals is set, the enabled thread is forwarded directly to the next node. If neither of the input threads is enabled, the comparator needs simply to keep the output valid-signal unset.

Additionally, three small control units are attached to each of the thread buffer blocks. The first determines whether the thread should be marked as enabled when scheduling ready threads, according to the status flags of the thread. The second is used to check if the thread is blocked by some semaphore that is just released, using the ID of the released semaphore and the ID of the needed semaphore that

is saved in the thread buffer block. If both values match, the enable signal of the thread is set. The BTC-PQ does not know what kind of threads are being scheduled (the ready or blocked ones) and only selects the highest-priority thread that is enabled for the current scheduling process. The third control unit is used to awake the thread if it is sleeping and the specified sleep time has elapsed. It decrements the register holding the absolute sleep time every clock cycle until it becomes zero, and then clears the SLEEPING-flag of the thread.

One of the major advantages of RTM is the sharing of resources among ready, blocked and sleeping threads. Another advantage is the acceleration of semaphore-related operations. Also, the ability of managing sleeping threads completely in hardware reduces the burden of the software part of a embedded system and provides more accurate timing delays. Unfortunately, this scheduler inherits the main problems of the BTC-PQ architecture, including the limited scalability and the lack of fairness among threads. Furthermore, this scheduler does not support handling interrupts.

A. Garcia et al. proposed an extended version of the BTC-PQ architecture to implement a real-time thread scheduler with support for complex deadline-based algorithms [37]. The extra logic for updating the priority of a thread is added into each of the thread buffer blocks directly. The nodes in the comparator tree are extended with an adder and a small opcode decoding logic. Due to the increased complexity of each node, the entire tree is pipelined to meet the timing requirement. Also, a central *finite state machine* (FSM) is employed to control and coordinate each part of the scheduler. An important advantage of this scheduler is integrating the interrupt handling directly into the hardware-based thread scheduling framework, which greatly reduces interrupt latency. The major problems of this scheduler include the unsolved bus loading problem and the hardware usage that is dramatically increased with the rising number of in-queue threads. Also, it does not support or accelerate any semaphore-related operations. Another drawback is the lack of the ability to manage sleeping threads.

Jamuth uses the BTC-PQ architecture for scheduling four hardware threads. From the architecture point of view, this scheduler is quite similar to the one described above, consisting of a information buffer for each hardware thread, a 2-level comparator tree as well as a central control logic. The information buffers hold the current priorities of the 4 threads based on which one of them is selected by the comparator tree. However, from the usage point of view, this scheduler differs thoroughly from any other thread scheduler discussed in this thesis. To better illustrate the differences, the processor architecture used by Jamuth is first described briefly below.

The two major models that enable hardware-level multithreading on a uniprocessor are *temporal multithreading* (TMT) and *simultaneous multithreading* (SMT) [112]. A TMT processor allows only a single thread to issue an instruction each clock cycle, while a SMT processor allows for issuing instructions from multiple threads in the same clock cycle. TMT is further categorized into *coarse-grained multithreading* (CGMT) and *interleaved multithreading* (IMT). On a CGMT processor, a thread runs continuously until it is blocked or an interrupt (e.g. caused by a system tick) occurs. Then, the processor must effectively context switch to another thread. This implies that only a single thread exits in the pipeline of the processor. Note that the CGMT model is used by the AMIDAR processor as well as all preexisting Java processors except Jamuth. In contrast to CGMT processors, an IMT processor issues an instruction from a different thread every clock cycle in a RR manner, i.e. the processor pipeline contains multiple threads at a time. Thus, cycle-to-cycle context switches need to be performed between pipeline

stages. The goal of IMT is to avoid stalls and pipeline bubbles caused by cache misses, pipeline hazards and branch mispredictions. Upon the occurrence of such an issue, the context of the processor can be switched from the current thread to another in one cycle, which improves processor throughput [35]. SMT can be considered as a variation of IMT, which is extended for the superscalar architecture. According to the observation that a single thread only has limited amount of instruction-level parallelism, SMT aims to exploit thread-level parallelism and to decrease the waste associated with unused issue slots. Both IMT and SMT processors must provide multiple register sets that hold the contexts (like PC values or fetched instructions) of different threads loaded into the pipeline at the same time.

Jamuth is a SMT Java processor with support for four hardware threads, i.e. instructions from four threads can be executed in any stage of the pipeline at a time. These threads share a common execute stage (EXE) and therefore must be scheduled within a single clock cycle. A quite compact real-time thread scheduler based on BTC-PQ is inserted between the *instruction fetch* (IF) and *instruction decode* (ID) stages. It needs to forward the next instruction of the thread with the highest priority to the ID stage on the fly. The BTC-PQ architecture is perfectly suitable for this purpose, since this unique scheduling scenario concerns priority only and does not require any temporal ordering of the four threads. Also, the depth of the comparator tree does not have negative influence on the single-cycle-delay constraint due to the small number of threads.

The scheduler supports two purely priority-based scheduling algorithms: EDF and *guaranteed percentage* (GP) [20]. GP assigns each thread a fixed number of clock cycles within a very small time-slice (e.g. 100 cycles). This number serves as priority directly and is decremented by 1 every clock cycle if the corresponding thread is being executed. After all priority values kept in the thread information buffers become 0, they will be reset to the initial values again. In this way, GP ensures that each thread is assigned a statical percentage of processor time over the whole lifetime of an application. The advantage of this algorithm is the absolute isolation of the 4 hardware threads so that no one can harm the timing constraints of any other.

In general, the SMT model used by Jamuth has several disadvantages. One of the major problems is the clearly increased complexity of the pipeline structure, which results in a long critical path that limits the clock frequency of the entire system. The performance of a single-threaded application could be degraded due to the lower frequency or an extra pipeline stage introduced to shorten the critical path. Also, only up to 2 real-time threads can be mapped directly to the hardware thread slots, because one of the slots must be reserved for running all non real-time threads and another for performing garbage collection [113].

Schedulers based on SR-PQ and SA-PQ

The real-time scheduler proposed in [62] contains two SR-PQ instances for keeping ready and sleeping threads respectively. It does not provide any semaphore-related operations and supports only the RM and EDF scheduling algorithms, using the ready thread queue. The RM algorithm does not allow the priority of a thread to be changed at runtime and the EDF algorithm updates the priority of a thread only directly before it is enqueued. Thus, both of them do not need to tackle the dynamic priority problem described above. The sleeping thread queue sorts all in-queue threads in reverse order of their awake time, i.e. the earlier the awake time, the higher the priority. Therefore, its PQ-head always holds the

thread that needs to be first awakened among all sleeping threads. The control logic of the scheduler must compare the priority of the PQ-head (i.e. the awake time) periodically and removes it from the queue if its sleep time is already over. However, the scalability and functionality of the scheduler is strictly limited by the size and number of the SR-PQ instances used in it. Also, this scheduler can only handle up to 8 interrupts.

The real-time scheduler proposed in [89] has three SA-PQ instances that hold ready, inactive and interrupt service threads respectively. Exploiting these queues, two deadline-based scheduling algorithms are implemented, namely EDF and LSF. As discussed above, such deadline-based scheduling algorithms update the priority of a thread only directly before the thread is re-enqueued. This means that the scheduler does not need to handle the situation that the priority of an arbitrary thread is changed at an arbitrary time point. For this reason, the lack of the ability of random access to any block in a SA-PQ does not affect the functionality of the scheduling algorithms used. The advantage of this scheduler is handling interrupts completely under the thread scheduling framework. One of the major weaknesses of this scheduler is that it cannot perform semaphore-related operations. Also, it does not support managing sleeping threads.

Schedulers based on FIFO-PQ

Currently, the *real-time unit* (RTU) [2] is the only commercial hardware-based thread scheduler, which is also known as *Sierra Kernel*. It aims to provide the key operations of a RTOS and supports 16 threads, 8 priority levels, 16 binary semaphores as well as 8 external interrupts. For the purpose of scheduling ready threads, a FIFO-PQ is employed [105]. This implies that the FIFO-PQ contains a total of 8 FIFO-queues each of which has 16 thread slots. Although the resource usage in this case is not unacceptable, no other FIFO-PQ is adopted in the scheduler.

To assist with executing semaphore-related operations, each semaphore is associated with a blocked thread queue of size 4. All threads in the queue are sorted by a control unit in decreasing order of priority, ensuring that the highest-priority thread can acquire the semaphore as soon as the semaphore is released by the current owner. The overflow of the queue is handled in software only. Also, all delayed (i.e. sleeping) threads are held in a separate queue architecture, which is checked sequentially in a fixed period. Once the delay time of a thread has expired, the thread is removed from the queue and added into the ready thread queue again.

Interrupt service threads are handled in a quite similar way like delayed threads. They are also held in a dedicated queue that includes an extra field in each slot to save the interrupt ID. Once an interrupt arrives, the interrupt handling unit traverses the queue sequentially and checks each thread according to its interrupt ID. After a corresponding IST has been found, the interrupt handling unit compares its priority with that of the current thread. Only if the IST found has a higher priority, the current thread may be preempted by it. This implies that the handling of an interrupt could be delayed arbitrarily long if priorities of some threads were missassigned.

RTU is the first attempt to implement an entire RTOS in hardware. It captures all key operations of an ordinary RTOS, from semaphore management to interrupt handling. However, its usage is limited to small embedded systems, due to the restricted number of supported threads, semaphores as well as interrupts. Also, it does not allow changing the priority of a thread at runtime.

Hthreads is another well-known hardware-based RTOS kernel [7]. It aims to provide an efficient programming model that crosses the FPGA-CPU boundary and abstracts away low-level details so that software programmers can apply their skills to hybrid systems containing both FPGA and CPU. To meet this goal, Hthreads maps FPGA-based computations to hardware threads and manages them along with CPU-based software threads under one common scheduling framework. Consequently, the differences between both computational types become transparent to programmers.

The hardware part of Hthreads consists of 4 key components: a thread manager, a thread scheduler, a mutex manager and an interrupt handler. Unlike other schedulers described above, Hthreads uses a distributed architecture and connects all its components independently to the peripheral bus and allows the CPU to access each of them through memory-mapped registers. Also, these components are implemented as both bus masters and slaves so that they can communicate autonomously with each other and with various FPGA-based accelerators that are also attached to the peripheral bus.

The thread scheduler employs the FIFO-PQ architecture to schedule ready threads. It solely holds the IDs of the head and tail threads of every FIFO-queue. A thread ID corresponds to an index into a central thread table. Each of the other threads in the FIFO-queue simply holds the IDs of the previous and next threads, which results in a doubly linked list. This variation of the FIFO-PQ architecture avoids the resource redundancy caused by multiple full-sized FIFO-queues at the expense of complications for the enqueue and dequeue operations. These two operations take 28 and 24 clock cycles respectively [4], regardless of the number of threads. Another advantage of this architecture is the ease of changing thread priorities, since a thread can be easily located in the central thread table via its ID.

Similar to the thread scheduler, the mutex manager also contains a thread table saving all blocked threads together, i.e. there is no separate queue for each mutex. All threads blocked by the same mutex are linked together in the FIFO order without concern for their priorities. As the mutex is released, the first thread in the linked list acquires the mutex and is added back to the ready thread queue.

Hthreads uses an active interrupt handling mechanism and starts all ISTs periodically. Once an IST is started, it first checks whether the interrupt that should be handled by it has occurred by sending the interrupt ID and its thread ID to the interrupt handler. The interrupt handler includes a blocked IST table and a pending buffer holding all unhandled interrupts. If the queried interrupt already exists in the pending buffer, the interrupt handler sets the VALID-flag in its status register. Otherwise, it inserts the ID of the IST into the blocked IST table. After reading the status register of the interrupt handler, the IST either begins handling the interrupt if the VALID-flag is set or sends a BLOCK command and its ID to the thread manager. Once a new interrupt arrives, the interrupt handler checks the blocked IST table to see if the corresponding IST has been blocked. If this is the case, it sends the ID of the IST to the thread manager so that the IST can be added back to the ready thread queue. After this IST is started again, it continues the interrupt service routine from the point where it was blocked.

3.5 Hardware Debuggers

Debuggers for modern soft-core processors aim to provide a convenient way to pinpoint the location of problems in an FPGA based SoC design. Such a debugger should enable a direct access to the FPGA, in order to allow the internal state of a SoC design to be extracted and analyzed at runtime. The most common and intuitive way to offer this ability is to insert extra hardware logic into the original design.

Important examples include *logic analyzer* like Vivado Debug Core [117] or ChipScope [126], *scan-chain* [40, 58, 110, 118] and *processor debug monitor* like MicroBlaze Debug Module (MDM) [70].

A logic analyzer can continuously sample several specific signals selected by a designer at synthesis time, using additional buffers and a trigger module. The values sampled are transferred through a JTAG connection to the debugging host and illustrated in terms of waveforms. In contrast to a logic analyzer, a scan-chain based debugger exploits the memory elements already built into a SoC design to shift the current values of these elements out of the FPGA, by inserting additional control logic in front of the memory elements. Both approaches conceptually enable monitoring the runtime state of the entire design including the processor and all peripherals, however, at the expense of notable resource overhead. Sometimes, this overhead is not even acceptable for debugging a large design, e.g. a multi-core system with a number of peripherals like PCI or Ethernet cores. Thus, using one of both approaches, a designer usually has to select a small set of signals previously. If new signals need to be added to the set, the whole system must be resynthesized. Another critical aspect of debugging a processor is the change in timing caused by those additional circuits.

A processor debug monitor is typically developed to assist with debugging a particular soft-core. Such a monitor is able to provide detailed internal information about the processor at runtime and can be seamlessly connected to a present debugging framework like GDB [39] over a JTAG port. Most debug monitors also support manipulating the execution of a program under the control of a user, such that the user can test particular portions of code to address bugs. This approach is very similar to *in-circuit emulation* (ICE) [103] that is widely used in many hard-core debuggers, except that an ICE module has already been integrated into the manufactured version of a hard-core processor whereas a debug monitor is optional for a soft-core processor. The primary problem with this approach is that it only provides a view on the system at the granularity of the processor, i.e. everything that can be accessed by the processor directly, such as internal registers or external memory, can be monitored and controlled by a user. However, a finer view granularity cannot be reached, e.g. the state of the internal registers of a peripheral cannot be obtained, by solely using the debug monitor.

SimXMD [119] is a simulation based debugger for MicroBlaze that connects GDB directly to ModelSim [72] instead of MDM. It supports a replay mode that allows a user to check the state of the processor at any previous time step. Since this debugger uses the trace port of MicroBlaze only, it provides the same view granularity as MDM does. The primary limitation of this debugger is that it cannot modify any memory element during debugging. Also, it inherits all of the drawbacks of a simulator, including high complexity of the construction of a simulation environment for a real world application, long duration of simulating a complex testcase as well as generation of huge log files.

To overcome the weaknesses in the traditional debug approaches, some debuggers utilize the readback support built into FPGAs such as Virtex series FPGAs from Xilinx, including gNOSIS [56] and NIFD [8]. The former is intended to be used as an automatic verification tool, rather than a classical debugger. It combines a running SoC design with a simulator, through the use of readback bitstream holding values of all registers in the design. The design is first simulated for a constant interval to produce reference data. Subsequently, the design runs on an FPGA for the same interval and gives the current state in terms of readback bitstream to the debugging host. Then, the information held in the bitstream is compared with the reference data. On a match, the current checkpoint will be saved and the whole process will

be repeated for another period to achieve the next checkpoint. Otherwise, the error will be reported and the verification will be interrupted. The primary drawback of this tool is the tight coupling of the hardware design and the simulation because the simulator is typically orders of magnitude slower than the hardware. This makes it the primary bottleneck of the whole system, i.e. the verification duration is only determined by the simulation that could run over weeks. Another problem of this tool is the use of additional readback and Ethernet modules, which is unnecessary since the JTAG based readback port does not require any extra hardware. Also, this tool does not support reading data back from BRAMs.

Unlike gNOSIS, NIFD fits into the standard definition of a soft-core debugger. It includes a classical GDB interface at the user's site and a break point controller along with a JTAGlink module at the FPGA's site. The JTAGlink module establishes a communication channel with the host to transfer debugging commands. No dedicated hardware module is used for readback. The major problem with this approach is that it requires clock-gating, as BRAMs are read back. This would affect the DRAM controller in an unexpected way and cause system crashes.

The debugger proposed in [47] also utilizes the readback feature of modern FPGAs. However, a bitstream is returned back to an extra MicroBlaze processor rather than the debugging host directly. This additional processor serves primarily as a debug monitor like MDM mentioned above and communicates with the debugging host over a serial port to transfer bitstreams and commands. This debugger is intended to be used to perform basic functional validation of an arbitrary hardware design like a SHA1 module instead of a soft-core processor only. Therefore, it does not provide any specific support for software debugging.

In addition, to the best of our knowledge, no readback based soft-core debugger supports any kind of monitoring external memory directly. Without this ability, it would be difficult to debug some part that is external to the FPGA, e.g. the heap of the AMIDAR processor. Section 4.7 describes the AMIDAR debugging framework [66] that allows extracting current state from any on-/off-chip memory elements, including registers, BRAMs and DRAM, which enables fine-grained debugging of a soft-core processor. However, a key thing to note is that monitoring DRAM is based on code injection instead of readback.

4 Implementation

4.1 Overview

Figure 4.1 illustrates the structure of a *system-on-chip* (SoC) that consists of an AMIDAR core and multiple peripherals. This SoC has been built for the evaluation purpose. Based on it, fundamental information on the AMIDAR processor is provided in the following.

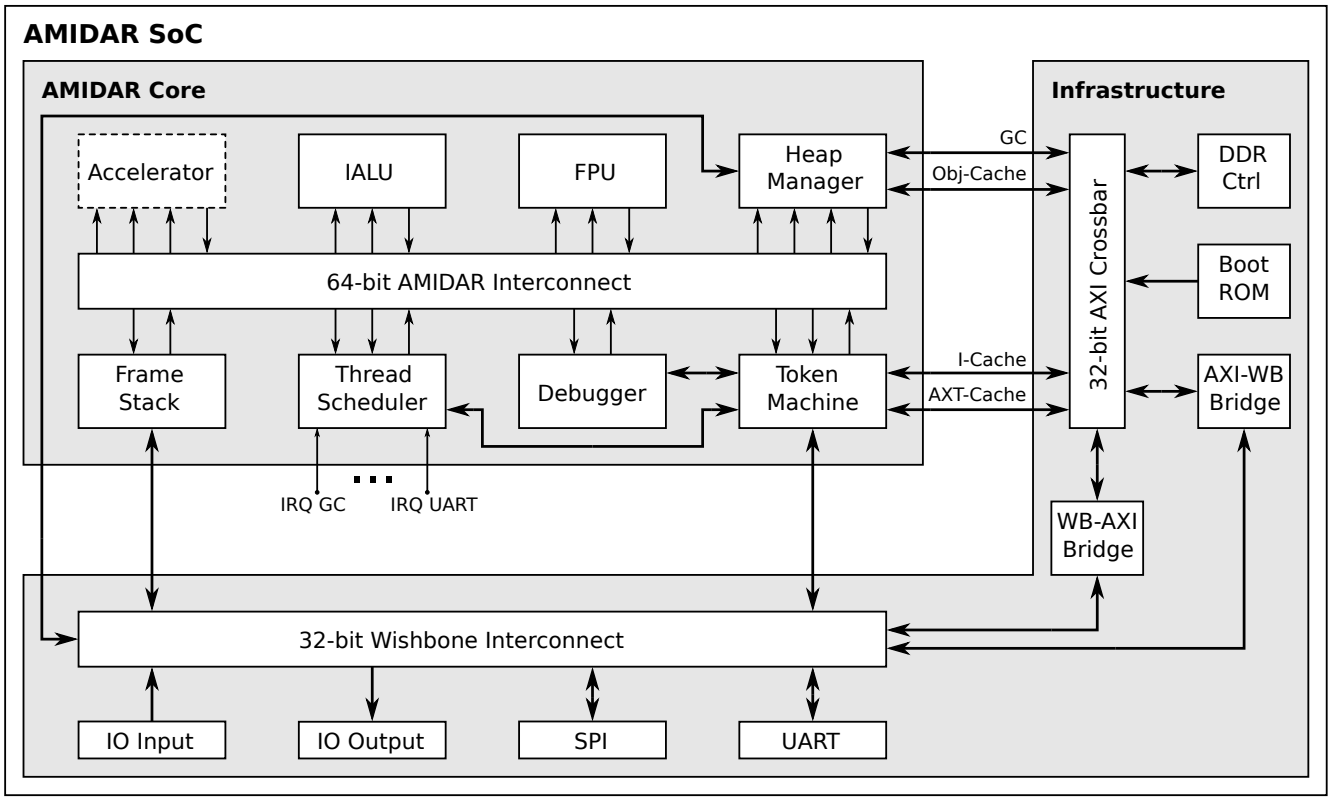


Figure 4.1: AMIDAR SoC

4.1.1 Processor Microarchitecture

The AMIDAR model allows each of its parts to be highly customized so that any instruction set architecture can be realized based on it. This is reflected by the microarchitecture of the AMIDAR processor, which results from the virtual architecture defined by the Java bytecode. This section describes each component of the AMIDAR processor briefly. Some of these components are explained in detail in the following sections of this chapter below.

Token Machine

The token machine plays a double role in the Java processor. On the one hand, it drives the whole system by generating and distributing tokens to different FUs. On the other hand, the token machine itself is also an FU and needs to execute tokens delivered to it. Therefore, the token machine has been implemented as a combination of a token distribution pipeline and a token execution module. The token distribution pipeline is composed of three stages, namely *fetch*, *decode* and *distribute*. The pipeline works

autonomously as long as bytecodes are executed in sequence. The token execution module interferes with it only upon the occurrence of one of the following cases: execution of a branch bytecode, thread context switch, exception handling and hardware-level debugging. Section 4.3 provides a thorough description of the token machine.

Frame Stack

The internal memory of the frame stack is partitioned into multiple Java stacks of equal size. Each thread alive is assigned a dedicated Java stack. Once a thread invokes a method, a new frame is pushed onto its stack, which consists of an operand stack, a local variable memory as well as a frame data section. The frame data section holds information about this frame and the caller of the current method. At a time, there can be only one active Java stack which belongs to the currently running thread. A stack pointer is employed to track the top address of the active stack. In addition to the stack memory, the frame stack also contains a thread table whose entries are indexed by thread identifiers. Each entry in this table saves the base address and stack pointer of some Java stack. The implementation of the frame stack is presented in Section 4.4.

Heap Manager

The entire heap of the AMIDAR processor resides in the external memory in order to provide sufficient space for holding all objects created by running an application. The heap manager implements the automatic management of the heap memory. For this purpose, it includes an object allocator and a mark-compact garbage collector. One of the key design goals of the garbage collector is to ease *direct memory access* (DMA) in the compact phase. To meet this goal, objects, especially, arrays used as the data buffers for DMA-capable peripherals may be locked explicitly so that the garbage collector can identify and skip them during the heap compaction. In this way, DMA can be safely performed at any time. Additionally, the heap manager also contains an object cache that is intended to facilitate accessing objects. This cache adapts a novel index generation scheme that can provide a better average miss rate than the classical XOR-based scheme described in Section 3.2. Every key component of the heap manager is described in Section 4.5.

Thread Scheduler

The thread scheduler realizes the preemptive scheduling model of Java. To this end, it contains a thread table, a system timer, a weighted round-robin arbiter (WRRA) as well as multiple thread queues. The thread table holds a number of attributes for each thread, like its priority or current state. The system timer generates a system tick upon the expiration of the current time-slice, which causes a thread context switch inside both token machine and frame stack. After that, the scheduler resets the system timer and assigns the newly started thread a time-slice according to its priority. Then, it selects the next thread from those saved in the ready thread queue, using the WRRA. Furthermore, the thread scheduler also includes a monitor table that is used to implement thread synchronization. Section 4.6 gives more details on the thread scheduler.

Debugger

The debugger is a simplified version of the token machine. It can communicate with the software part of the AMIDAR debugging framework via a *Joint Test Action Group* (JTAG) port. Whenever the token machine is halted due to some debugging-specific event like the occurrence of a breakpoint, the debugger takes over its role and controls the AMIDAR processor according to the commands received over the JTAG port. It translates these commands into regular tokens, distributes them to the other FUs and then returns the results through the JTAG port as well. In Section 4.7, the AMIDAR debugging framework is introduced.

Integer ALU and Floating Point Unit

The IALU and FPU execute arithmetic and logical operations and are compatible with the Java specification. Both are used as separate functional units, increasing the parallelisms of the entire system. Since these FUs solely implement the common integer and floating-point operations, this thesis does not discuss them further.

Token Distribution Network

The token distribution network is constructed around the token machine, using a star topology. For each FU, the token machine provides a dedicated token distribution port that is connected to the FIFO-based token buffer of the FU directly. If the token buffer of any FU becomes full and therefore cannot receive a new token, the token machine will suspend itself and wait until the current token of the FU has been executed.

Data Bus

The FUs of the AMIDAR processor communicate with each other over a point-to-point interconnect that consists of multiple directed connections. With the help of an analysis tool, these connections are extracted from the ADLA description of the Java bytecode automatically. Each of them links the output port of an FU to some input port of another FU. Multiple connections to the same input port of an FU are multiplexed. If more than one data packet is delivered via these connections, only the one with the tag of the current token of the FU is selected and passed on further to the input port. In comparison with a classical shared bus architecture, the point-to-point interconnect guarantees that any data packet can be sent to its destination FU without unnecessary delay.

4.1.2 Support for 64-Bit Operations

The Java bytecode includes a number of arithmetic and logical instructions operating on 64-bit operands like **`ladd`** (addition of two values of type **`long`**). To speed up executing such instructions, the AMIDAR processor provides architectural support for them in the following two ways. First, the IALU and FPU implement all 64-bit operations defined in the Java Virtual Machine specification [50]. Second, the point-to-point interconnect as well as the input and output ports of every FU are all 64-bit wide, which allows any operand or the result of a 64-bit operation to be transported in a single transfer cycle. Each port of an FU, whether input or output, are associated with two 32-bit data registers that are employed

to buffer the low and high parts of a 64-bit value. To perform 32-bit operations, only the register that is intended to buffer the low 32 bits is actually used and the other one is simply ignored by the FU.

The token sets of the instructions operating on **long** or **double** operands are considerably simplified through the 64-bit support built into the AMIDAR processor. For example, Listing 7 illustrates the token set of **ladd**, which contains as many tokens as that of **iadd** described in Section 2.1.2.

Listing 7: Token set of ladd

```
0: ladd
1: {
2:   T(framestack, POP64, ialu.1),
3:   T(framestack, POP64, ialu.0),
4:   T(ialu,       LADD,  framestack.0)++;
5:   T(framestack, PUSH64)
6: }
```

4.1.3 Infrastructure

The AMIDAR processor adopts a hybrid infrastructure that is made up of a *Wishbone* (WB) interconnect and an *Advanced eXtensible Interface* (AXI) crossbar. The WB interconnect is used to attach peripherals to the AMIDAR core so that a number of open source IP cores with the WB slave interface can be easily built into AMIDAR-based SoCs. The AXI crossbar is required for the connection of the AMIDAR core to the external memory (i.e. DRAM). This is because the memory controller provided for the Artix-7 FPGA that was chosen as the development platform for the AMIDAR processor supports the AXI protocol only.

At the software level, every peripheral is represented in the form of an object and each of its registers maps to an integer field of the object. Such an object is referred to as a *WB object* below. It differs from a regular object by its handle whose highest bit is set to 1. Any access to a WB object is redirected over an AXI-to-WB bridge to the corresponding peripheral, which is very close in spirit to the classical memory-mapped I/O. Optionally, an FU may also be connected to the AMIDAR core via an additional WB slave interface so that it can be accessed through a WB object at runtime. This allows the FU to be initialized, monitored and reconfigured dynamically, increasing the flexibility and usability of the AMIDAR processor. Section 4.5.6 describes the realization of WB objects in more detail.

Besides WB objects, another two communication approaches between the AMIDAR core and peripherals have been implemented as well, namely hardware interrupts and DMA. The interrupt handling mechanism of the AMIDAR processor is thread-based, as mentioned in Section 2.2.2. Thus, the *interrupt request* (IRQ) signals of all interrupt-capable peripherals are connected to the thread scheduler directly. Upon the assertion of any IRQ signal, the corresponding interrupt service thread is started accordingly.

In stead of the classical *third-party* DMA performed by a dedicated controller, the AMIDAR processor supports the more efficient *bus mastering* DMA. Due to this, any DMA-capable peripheral needs to implement the WB master interface in addition so that it can initiate DMA transactions autonomously. Since the external memory is connected to the AXI crossbar rather than the WB interconnect, a WB-to-AXI bridge has been introduced to enable direct data transfers between the memory and peripherals.

To assist with customizing an AMIDAR-based SoC, a graphical tool called *system builder* has been developed. Using it, different peripherals can be easily added to or removed from the target SoC. Once the customization is complete, the system builder will generate the source code for the infrastructure and the glue logic that connects the AMIDAR core, the selected peripherals and the infrastructure together. Additionally, the system builder also assigns every FU and peripheral included in the SoC a unique handle. At synthesis time, the handles of all WB objects are written together with the pregenerated bytecode stream of the bootstrap method into a BRAM-base read-only memory called *Boot-ROM*.

4.1.4 Native Methods

A native method does not contain any concrete implementation and typically needs to be realized by using a platform native language like C. Since Java is already the native language of the AMIDAR processor, most of preexisting native methods have been rewritten in Java. The remaining ones are those that carry out specific functions which otherwise cannot be implemented in software. Formally, such a method can be defined as a sequence of operations that are performed by different FUs in a fixed order. This is basically just the way how a bytecode is executed on the AMIDAR processor. Therefore, the most straightforward approach to implementing a native method is to program it with tokens. Currently, a total of 6 native methods have been implemented in this way and they are all declared in class **de.amidar.AmidarSystem**. The function of each of these methods is described briefly below:

- **public static native int readAddress (int addr)**

  This method reads out the 32-bit value stored in the location addressed by **addr** and returns this value as an integer. Not just the data in the external memory can be accessed by using this method, but also those held in the Boot-ROM and peripheral registers.

- **public static native void writeAddress (int addr, int value)**

  This method performs the inverse operation of the one above and writes the given integer **value** to the location addressed by **addr**. If **addr** falls within the address range of the Boot-ROM, nothing will happen.

- **public static native Object intToRef (int value)**

  This method converts the given integer **value** to an object handle. This kind of type casting is required, e.g. to convert the preassigned handle of a WB object that is saved in the Boot-ROM as an integer to the actual handle of the object.

- **public static native int refToInt (Object ref)**

  This method converts the given object handle **ref** to an integer. It is utilized, e.g. to generate the hash code of an object.

- **public static native void flushRef (Object ref)**

  This method flushes all cache blocks holding the object referenced by **ref**. It is intended to write the cached data buffer of a DMA-capable peripheral back to the external memory, before a new DMA transaction is initiated.

- **`public static native int gcLock(Object ref)`**

  This method locks the object referenced by **`ref`** so that the garbage collector will not reallocate it in the compact phase.

Each of the native methods above is assigned an unused bytecode. Upon converting the class files of a Java application into an executable file for the AMIDAR processor, the invocation of any of these methods is replaced with the corresponding bytecode. As a result, the token set defined for a native method will be executed, once the bytecode assigned to the method is encountered at runtime. Realizing native methods in this way is simple, but has a limitation on the amount of native methods allowed, because only 51 bytecodes are still free. To overcome this limitation, another approach that enables multiple native methods to be mapped to a single bytecode was designed, in particular, for the thread scheduler. Based on it, 14 thread- and synchronization-specific native methods have been implemented by using only two bytecodes. This approach is referred to as *functional unit native interface* (FU-NI) and is discussed in Section 4.6.2 in detail.

4.1.5 Executable Generation

A compact executable format has been designed for the AMIDAR processor. Section 4.2 below provides a thorough description of it. An executable file generated in this format is made up of all classes required for running a Java application. All these classes share a single constant pool, which greatly reduces the redundant information that is present in the original class files. During the generation of the executable file from the class files, the bytecode streams of different methods included in these files are checked by an analysis tool in addition. This tool patches specific bytecodes under certain circumstances as described below:

- Use of **`lookupswitch`** or **`tableswitch`**

  Both bytecodes above are the only ones that the AMIDAR processor does not support directly. Therefore, they need to be replaced with an **`if_icmpeq`**-chain. This patch has the same effect as if the corresponding **`switch`**-statement was replaced with a chain of **`if`**-**`else`** statements at the Java level.

- Invocation of a native method

  As mentioned above, each remaining native method has an associated bytecode. Once the analysis tool encounters an **`invoke`**-bytecode that calls a native method, the **`invoke`**-bytecode is replaced with the one associated with the native method.

- Occurrence of a synchronized method

  If the bytecode stream of a synchronized method is encountered, the analysis tool encloses the whole bytecode stream within a pair of **`monitorenter`** and **`monitorexit`**. As a result, the AMIDAR processor does not need to check the access flag of the method at runtime. This patch is also adopted by picoJava-II and JOP.

4.1.6 System Boot

After a system reset, the AMIDAR processor boots automatically as follows:

1. It starts executing the bootstrap method stored at a fixed position in the Boot-ROM. This method first loads the AXT file into the external memory via a UART that is included in every AMIDAR-based SoC by default. Then, it initializes the token machine, the frame stack and the heap manager according to the information held in the AXT file, like the base address of the heap.

2. After the three FUs have been initialized, the AMIDAR processor starts executing the bootloader method contained in the AXT file, which performs three tasks. First, it invokes the static initializers of all classes in a predefined order. Second, it reads the handle of every WB object from the Boot-ROM as an integer and converts this value to a handle by using the **`intToRef`**-method. If the corresponding peripheral is interrupt-capable, it creates an interrupt service thread for the peripheral in addition. Last, it calls the **`main`**-method of the application.

4.2 AMIDAR Executable Format

Unlike a JVM, the AMIDAR processor does not execute class- or JAR-files directly. A Java application first needs to be converted to an *AMIDAR executable* (AXT) file, before it can run on the AMIDAR processor. Since the AMIDAR processor is constrained in terms of both memory and speed, it heavily depends on having an executable format with the following qualities:

- Executable files based on this format must be as compact as possible.
- Executable files based on this format must be able to be executed efficiently.

To meet the former goal, all class- or JAR-files of an application are packed into a single file that only contains one shared constant pool[2]. This eliminates the redundant information that is present in the class files. Note that not just the classes which are explicitly defined for the application but also those recursively referenced API classes are included in the file, avoiding the need for dynamic linking at runtime.

To increase the execution performance, the classes contained in the file are resolved statically. For this purpose, the most fundamental information about these classes as well as their methods and fields is extracted from the original class files and stored in several information tables separately. Consequently, the symbolic references included in bytecodes can be replaced with the corresponding indexes into the information tables, which avoids the need for dynamic resolution at runtime. Due to this reconstruction, all class- , method- and field-related entries in the constant pool are not necessary for execution of the application anymore and therefore deleted completely, which reduces the size of the AXT file further. In the following, we describe the layout and content of an AXT file in detail.

4.2.1 Layout

An AXT file is composed of four major parts: a *header*, an *info section*, a *table section* as well as a *data section*. A brief overview on these parts is given below.

[2] This idea came from the *dalvik executable* (DEX) format [29].

- ***Header***: The header provides primary information on this AXT file, like the offset of each of the following parts.
- ***Info section***: This section currently holds auxiliary information for the garbage collector only.
- ***Table section***: This section can be divided into several subsections further, including a *class table* (CT), a *method table section* (MTS), a *static method table* (SMT), an *exception table section* (ETS), an *implemented interfaces section* (IIS) as well as an *interface table section* (ITS).
- ***Data section***: This section contains the shared constant pool, the bytecodes of each method, several pregenerated objects as well as a small handle table holding an entry for each of these objects.

The entire file is loaded into the main memory of the AMIDAR processor during booting. Except the data section, the rest of the AXT file remains unchanged throughout the whole lifetime of the application.

Each of these four parts is described below. To simplify the description, the info section is explained after the table section, because it needs to reference several data held in the table section.

4.2.2 Header

Although the basic layout of every AXT file is identical, the position and size of each part can be different from one application to another. Therefore, the header holds a number of application-specific information about the current AXT file as shown in Table 4.1, where all offsets are relative to the start of the file.

Name	Width	Description
`Magic`	32-bit	Magic value.
`Info_section_off`	32-bit	Offset to the info section in bytes.
`CT_off`	32-bit	Offset to the class table in bytes.
`MTS_off`	32-bit	Offset to the method table section in bytes.
`SMT_off`	32-bit	Offset to the static method table in bytes.
`ETS_off`	32-bit	Offset to the exception table section in bytes.
`IIS_off`	32-bit	Offset to the implemented interfaces section in bytes.
`ITS_off`	32-bit	Offset to the interface table section in bytes.
`Constant_pool_off`	32-bit	Offset to the constant pool in bytes.
`Code_section_off`	32-bit	Offset to the code section in bytes.
`Heap_off`	32-bit	Offset to the pregenerated object section in bytes.
`Handle_table_off`	32-bit	Offset to the handle table in bytes.
`Main_method_amti`	32-bit	Absolute method table index of the **`main`**-method.
`Run_wrapper_amti`	32-bit	Absolute method table index of the **`runWrapper`**-method.
`Interfaces_idx`	16-bit	Index of the first interface inside the class table.
`Array_types_idx`	16-bit	Index of the first array type inside the class table.

Table 4.1: Header of an AXT file

Since most entries shown in the table are self-explaining, we only describe the last three index values briefly.

Every class included in the AXT file has a dedicated method table holding all of its instance methods. In contrast, the static methods of all classes are stored together in a common table, i.e. the SMT. These tables are placed consecutively inside the AXT file and therefore make up a large table with all methods implicitly. The index of a method into this table is called the *absolute method table index* (AMTI) of the method. An arbitrary method can be addressed through its AMTI directly. In addition, every instance method of a class also has a *relative method table index* (RMTI) which represents its local position inside the method table of the class. The AMTI of the first method in a method table is referred to as the index of the method table (IMT). Given the RMTI of an instance method, M, and the corresponding IMT, the AMTI of the method can be calculated as follows: $AMTI_M = IMT_M + RMTI_M$.

The **runWrapper**-method is a static method that serves as the common entry point for all threads except the main one. Through the AMTI of this method, every newly started thread can begin running its task from the **runWrapper**-method in which its own **run**-method is invoked. This simplifies the thread management so that we do not need to know the RMTI of the **run**-method of every thread instance (for more details, see Section 4.6.2).

In Java, both interfaces and arrays are treated as classes, i.e. they need to be saved in the class table. To provide a more structural layout, the class table is partitioned into three consecutive areas. The first area starts from the beginning of the table and holds all regular classes. The two following areas include interfaces and array types respectively. Thus, through the last two indexes held in the header, both latter areas of the class table can be located.

According to the information provided by the header, the bootloader can identify each part of an AXT file and therefore load them into the main memory of the AMIDAR processor properly.

4.2.3 Table Section

The table section is composed of a number of tables that expose primary information about all classes as well as their methods and fields to the AMIDAR processor (in particular, the token machine). These tables are further partitioned into several subsections as mentioned above. We explain each of these subsections separately below.

Class table

For each class included in the AXT file, there is an entry in this table, whose index is referred to as the *class table index* (CTI) below. The first entry is reserved for class **java.lang.Object**, i.e. its CTI is always equal to 0.

As mentioned above, all entries in the class table can be divided into three parts: regular classes are stored at the beginning of the table, followed by interfaces and then array types. Since all interfaces are saved in succession, they form an *ordered list of interfaces* (OLI) implicitly. Thus, an interface can also be referenced by its index in the OLI (IOLI) in addition to its CTI. The relationship between both indexes of an interface, I, can be formally represented as such: $CTI_I = Interfaces\_idx + IOLI_I$.

An array type differs from a normal class in two ways. First, it does not have an explicit definition and is solely identified by its element type as well as the number of its dimensions. Second, it inherits all methods from class **java.lang.Object** and implements interface **java.lang.Cloneable** by default. All array types are saved at the end of the class table and sorted as follows:

- Array types with primitive element types are first inserted into the class table before those with reference element types.
- Array types with the same element type are inserted into the class table successively in ascending order of their dimension numbers.

Just like the interfaces, the array types make up together an *ordered list of of array types* (OLAT). Therefore, an array type, A, can also be identified by its index in the OLAT (IOLAT). The IOLAT of A can be converted to the CTI of A in the following way: $CTI_A = Array\_types\_idx + IOLAT_A$.

Tables 4.2 and 4.3 demonstrate the attributes of a class and an array type respectively. As shown in the tables, both of them have a total of six attributes, where the second and fifth attributes are redefined for an array type. We give a brief explanation about each of these attributes in the following.

Name	Width	Description
`Flags`	16-bit	Flags of the class.
`Obj_size`	16-bit	Size of an object created from the class in bytes.
`Super_idx`	16-bit	CTI of the super class.
`IIS_bitmap_off`	16-bit	Offset to the corresponding bitmap in the IIS in bytes.
`ITS_idx`	16-bit	Index of the corresponding entry in the ITS.
`MT_idx`	16-bit	IMT of the the method table of the class.

Table 4.2: Attributes of a class

Name	Width	Description
`Flags`	16-bit	Flags of the array type.
`Element_type`	16-bit	Array element type.
`Super_idx`	16-bit	0 (**`java.lang.Object`**).
`IIS_bitmap_off`	16-bit	Offset to the corresponding bitmap in the IIS in bytes.
`Dimension_num`	16-bit	Number of the array dimensions
`MT_idx`	16-bit	IMT of the the method table of **`java.lang.Object`**.

Table 4.3: Attributes of an array type

Flags: Currently, only the two least significant bits of this attribute are used, namely $flags[0]$ and $flags[1]$. If $flags[0]$ is set, the current entry belongs to an array type; otherwise, this entry represents a class. For an array type, the assertion of $flags[1]$ indicates that its array element type is primitive; otherwise, reference.

Obj_size/Element_type: If the current entry corresponds to a class, this attribute provides the size of an object created from the class in bytes. If the entry belongs to an array type whose element type is reference, this attribute holds the CTI of the reference type; otherwise, an integer in range between 0 and 7, which represents one of the eight primitive data types defined in Java.

Super_idx: This attribute saves the CTI of the super class of the current entry. As mentioned above, the super class of an array type is **`java.lang.Object`** whose CTI is equal to 0. Thus, this attribute of an array type is always set to 0.

IIS_bitmap_off: The implemented interfaces section contains a number of bitmaps of same size. Each bit of a bitmap corresponds to an interface contained in the OLI. A class is assigned a bitmap in this section only if it implements at least one interface. Different classes may share a single bitmap if they implement same interfaces. This implies that all array types have one common bitmap in which only the bit that represents **java.lang.Cloneable** is set. The size of a bitmap is application-dependent. For this reason, a bitmap is referenced by the offset from the start of the IIS to it rather than an index. The first bitmap (i.e. the 0-th) is reserved for classes that implement no interface and therefore all bits in it are set to 0. This implies that `IIS_bitmap_off` of a class is equal to 0, if it does implement any interface.

ITS_idx/Dimension_num: If a class implements at least an interface, i.e. it has a bitmap in the IIS, an extra interface table is created in the ITS for it. This table has as many entries as the OLI. However, an entry in the table is valid only if the corresponding bit in the bitmap of the class is set, i.e. the class actually implements the interface represented by the asserted bit in the bitmap. In this case, the entry saves the RMTI of the first method declared in the interface. All methods of the interface are placed successively in the method table of the class in their declaration order inside the interface. Thus, each of them can be easily addressed according to the RMTI provided by the interface table and its declaration order. The IIS and ITS as well as the invoking mechanism of an interface method are described in more detail below. For an array type, the number of its dimensions is saved in this attribute. As mentioned above, every array type implements interface **java.lang.Cloneable** by default. However, this interface does not contain any method declaration. Thus, this attribute may be redefined for an array type.

MT_idx: This attribute holds the AMTI of the first instance method in the method table of the current entry, i.e. the IMT of this entry. If the current entry belongs to an array type, this attribute is set to the IMT of class **java.lang.Object**, allowing the array type to inherit all methods from the **Object**-class.

Method Table Section and Static Method Table

The MTS contains a method table for each class included in the AXT file. Every non-static method of a class such as an instance method or a constructor has an entry in the method table of the class. This implies that all methods of the class which need to be invoked via bytecodes **invokevirtual** and **invokespecial** are stored in this table.

A key thing to note is that the method table of a class does not just hold the non-static methods declared directly in the class but also those of its super class. Therefore, during construction of the method table of a class, the entire method table of the direct super class of this class needs to be copied to the beginning of the newly created method table at first. After that, the non-static methods of the class are written into the table in the following order: public, protected, package private and then private methods. If some method overrides the corresponding method inherited from the super class, it simply takes the place of the inherited one. As a result, this method has the same RMTI in the method tables of the current class and its super class, which realizes polymorphism of Java automatically. Also, this greatly simplifies the execution of bytecode **invokevirtual** on the AMIDAR processor, because every instance method can be located in the MTS in constant time by exploiting its RMTI.

If a class implements an interface, all methods of the interface are saved consecutively in the method table of the class in their declaration order. Therefore, given the RMTI of the first method declared in

the interface, the relative indexes of the other methods of this interface can also be determined easily. Such a method can be invoked via either **invokevirtual** or **invokeinterface** on an object of the class, depending on the type of the reference to the object, through which the method has been invoked. In the former case, the method can be addressed by its RMTI as described above, which is impossible in the latter case. This is because multiple classes can implement the same interface, each of which can also implement a number of different interfaces in addition. Therefore, it is not possible to write an interface method to a fixed position in the method table of every class that implements the corresponding interface. The approach used to solve this problem is discussed later in this section below.

A static method is unique in a Java application and does not provide polymorphism. Therefore, the static methods of all classes are held separately in the SMT which follows directly behind the MTS. Every static method can be found simply through its AMTI, making the execution of **invokestatic** very efficient. Note that the SMT and a method table included in the MTS contain the same attributes for describing a method, which are shown in Table 4.4.

Name	Width	Description
`Flags`	8-bit	*Currently unused.*
`Num_arg`	8-bit	Number of the arguments.
`Max_stack`	16-bit	Maximum depth of the operand stack .
`Max_locals`	16-bit	Maximum number of the local variables.
`Exception_table_size`	16-bit	Size of the exception table
`Exception_table_idx`	16-bit	Index of the exception table.
`Code_length`	16-bit	Length of the bytecode stream in bytes.
`Code_off`	32-bit	Offset to the bytecode stream inside the code section in bytes.

Table 4.4: Attributes of a method

Exception Table Section

Each method with a **try-catch** block is associated with an exception table in the ETS, which provides the following information on every exception handler of the method as shown in Table 4.5, where a PC is given in terms of an offset relative to the start of the bytecode stream of the method in bytes.

Name	Width	Description
`Start_PC`	16-bit	Start of the bytecode range in which the exception handler is active.
`End_PC`	16-bit	End of the bytecode range in which the exception handler is active.
`Handler_PC`	16-bit	Start of the exception handler.
`Catch_type`	16-bit	CTI of the exception type.

Table 4.5: Attributes of an exception handler

Just like the method tables in the MTS, all exception tables are arranged successively in the ETS and therefore can be considered as a large table holding all exception handlers. For this reason, attribute `Exception_table_idx` of a method actually represents the index of the first exception handler of the method in this implicit table. Attribute `Exception_table_size` of a method provides the number of entries in the exception table of the method, which is necessary for the exception handling process. This

is because, when an exception is thrown during execution of a method, the token machine needs to traverses the exception table of the method to find the corresponding handler. For this purpose, the size of the exception table is required.

Implemented Interfaces Section

In Java, a class may implement arbitrarily many interfaces. To facilitate executing bytecodes **`instanceof`** and **`checkcast`**, every class that implements at least an interface is assigned a bitmap in the IIS. The n-th bit, B_n, in each bitmap is associated with the interface I whose IOLI is equal to n. If a class implements I, B_n in its bitmap is set to 1. Besides the bits corresponding to the direct interfaces of a class, those that represent the super interfaces of these interfaces as well as the implemented interfaces of the super class are also asserted in the bitmap of the class. Classes that implement no interface share a common bitmap, namely the 0-th, in which all bits are set to 0. To ensure a bitmap to be byte-aligned, several padding bits can be added to it.

Interface Table Section

Each class that implements at least an interface is assigned an interface table in the ITS, which has as many entries as the OLI. This table allows a method declared in the interface to be located in constant time. To better illustrate this, we first explain the way how a method is executed on the AMIDAR processor in general.

As mentioned above, several bytecodes are statically resolved by exploiting the information tables defined in the AXT format to increase execution performance. For those bytecodes that are used to invoke methods such as **`invokevirtual`** or **`invokeinterface`**, the primary goal of the static resolution is to facilitate determining the AMTI of a given method as efficiently as possible.

For example, the original operand of bytecode **`invokevirtual`** is a reference to a constant pool entry that describes the method which needs to be invoked symbolically. During the static resolution, this symbolic reference is replaced with the method argument number and the RMTI of the corresponding method. Upon invoking an instance method on an object, the token machine first reads the CTI of the class of this object from the handle table. Then, the value of attribute `MT_idx` (i.e. the IMT) is added to the RMTI, resulting in the AMTI of the method. Using the AMTI, the token machine fetches the first bytecode of the method and starts executing it.

However, for bytecode **`invokeinterface`**, its symbolic reference cannot be replaced with a fixed RMTI simply. The reason is that a method of an interface is assigned different indexes in the method tables of various classes that implement this interface, as discussed above. To solve this problem, the symbolic reference of **`invokeinterface`** is replaced with the IOLI of the interface and the declaration order of the invoked method. Additionally, the RMTI of the first declared method of the interface is written into the interface table of every class that implements this interface. The entry holding this value is addressed using the IOLI of the interface. When an interface method is invoked on an object via **`invokeinterface`**, the token machine first obtains the corresponding CTI from the handle-table. Then, according to the value of `ITS_idx` held in the class table, the interface table of the class is found. Using the IOLI of the interface, the RMTI of the first method of the interface can be read out. Adding this RMTI to the declaration order of the invoked method results in the RMTI of the invoked method. Finally, the

AMTI of this method can be calculated by adding the RMTI to the value of `MT_idx` stored in the class table.

An interface table of a class consists of as many entries as the OLI. Each entry can hold a 16-bit RMTI. The value of the *n*-th entry is valid only if the class implements the *n*-th interface in the OLI. According to the measurements of real-world benchmarks, most classes that are assigned an interface table in the ITS implement only a small number of interfaces among those included in the OLI. Therefore, many entries of an interface table remain unused in most cases. To minimize the size of an AXT file, multiple interface tables with few valid entries are merged together, as demonstrated in Figure 4.2.

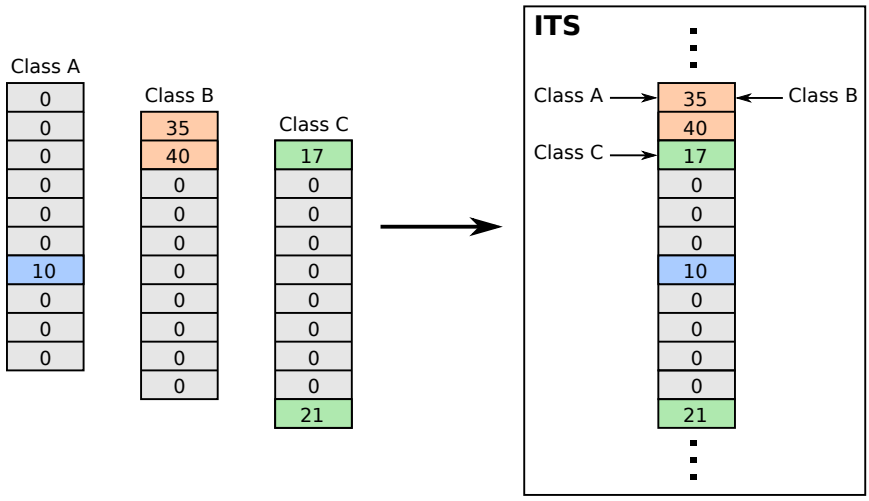


Figure 4.2: Merging multiple interface tables

4.2.4 Info Section

The info section is intended to assist with garbage collection, which consists of the following three parts: a *header*, a *class info table* and a *field info section*. Each of these parts is presented in detail below.

Header

The header provides the size of the info section in bytes and the RMTI of the **`finalize`**-method of class **`java.lang.Object`**. The former information is required for initializing the garbage collector so that the info section can be loaded into an internal module of the garbage collector properly. The latter one is used by the garbage collector to determine whether the **`finalize`**-method needs to be called on a dead object before it is deleted. Due to the way how the method table of a class is constructed, the **`finalize`**-method is located at the same position in every method table, i.e. its RMTI is a constant value for all classes included in the AXT file.

Class info table

Each 24-bit entry in this table provides the GC-specific information about a class. The two most significant bits of the entry are referred to as *finalize empty flag* (FEF) and *no reference field flag* (NRFF) respectively below. The former flag indicates if this class has an empty **`finalize`**-method and the latter whether it contains at least a reference field. The next three bits are called *reference type flags* and denote whether this class is one of the subclasses of **`Reference`**, namely **`SoftReference`**, **`WeakReference`** and

PhantomReference. If any of these flags is set, the class is either the corresponding subclass of **Reference** or derived from this subclass. The following three bits are reserved for future extension and currently serve as padding bits. The remaining sixteen bits represent a byte-aligned offset inside the field info section. This offset is valid only if the NRFF is not set. In the following, we discuss this information in more detail.

The **finalize**-method is an empty method defined in class **java.lang.Object**, which implies that all classes inherit it by default. According to the Java specification [49], this method must be invoked on an unreachable object before the memory of the object is reclaimed. However, if a class does not override the original definition of the method, calling it on an object of the class is actually unnecessary, since no action will be taken indeed. Based on this observation, all classes that do not redefine the **finalize**-method are marked using the FEF. Dead objects of these classes are removed by the garbage collector directly, without calling the **finalize**-method on them.

One of the major tasks of the garbage collector is to determine the reachability of every object from the root set. The root set is composed of all objects that are directly accessible to the program and therefore always reachable. Any object that is referenced by a field of a reachable object is also considered reachable. An object that is reachable from the root set is said to be live. The ability to distinguish the reference fields of an object from the primitive ones is essential for the garbage collector so that it can trace out the graph of references from the root set. To achieve this goal, every class with at least a reference field is assigned a bitmap. Each field of the class is mapped to a single bit, except those of type **long** or **double** which are represented using two consecutive bits. If the type of a field is reference, the bit related to it is set to 1. All bitmaps are stored in the field info section successively. To allow each of them to be byte-addressable, a bitmap may contain several padding bits in addition. An optional task of the garbage collector in the mark phase is to recognize every instance of type **Reference** and assign its referent a reachability level. This can be easily achieved, by exploiting the three reference type flags mentioned above.

Field info section

This section consists of a number bitmaps each of which is associated with a class except the first one (i.e. the one whose offset is equal to 0). This bitmap belongs to a virtual object called *static field object*. This object does not have a corresponding class and is assembled using the static fields of all classes. Each bit asserted in this bitmap represents a static reference field. In the mark phase of a garbage collection cycle, the static field object is treated just like a regular object and provides a important subset of the root set.

4.2.5 Data Section

This section can be partitioned into three major parts: a *constant pool*, a *code section* and an *immortal heap*. Since the code section simply stores the bytecode streams of all methods, the following discussion is solely focused on the other two parts.

Constant pool

As mentioned above, all classes included in the AXT file share a single constant pool. To minimize the size of this common constant pool, redundant entries like duplicated string literals are eliminated

during the aggregation of the constant pools contained in the original class files. Remaining string literals are converted to string objects and moved from the constant pool to the immortal heap. Due to the static resolution of bytecodes, entries that provide symbolic information about classes, methods and fields are not required anymore and therefore removed completely. Consequently, only those entries holding numeric constants still remain in the constant pool, including `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_Long` and `CONSTANT_Double`. As a bytecode whose parameter is one of these four constant pool entries is being resolved, e.g. **`ldc`**, the type of the entry is checked according to its tag in addition. If no type mismatch is found, the tag is deleted from the entry, because it is unnecessary to check the type correctness at runtime again. As a result, there are exclusively constant numbers in the aggregated constant pool after the static resolution.

Immortal heap

During the construction of the AXT file, a number of objects are pregenerated, including the static field object, multiple string objects and the **`char`**-arrays referenced by them, as well as a **`Class`**-object for every entry in the class table. The reasons for the pregeneration of these objects are discussed briefly below.

- The static reference fields are one of the important parts of the root set. Encapsulating these fields in a regular object allows the garbage collector to transverse them in a simple and general way during the mark phase, increasing performance and avoiding special cases.
- In a JVM, string literals are transformed to string objects as they are accessed. To reduce runtime overhead, these transformations are brought forward before runtime. This also eases the implementation of the token machine.
- A **`Class`**-object is traditionally created as the corresponding class file is loaded into the runtime system for the first time. Since all classes that might be required by the application are loaded and linked in the AXT file statically, their **`Class`**-objects should be generated in the meantime accordingly.

An important thing to note about the objects described above is that they will not be removed by the garbage collector throughout the entire lifetime of the application. For this reason, they are referred to as immortal objects below. During booting, these objects are loaded into a specific area in the main memory of the AMIDAR processor, which is called immortal heap. In addition to this area, there is also a dynamic heap area, which holds objects created at runtime.

Due to the indirect object addressing scheme used in the AMIDAR processor, all objects need to be accessed through a handle table, including the immortal ones. The handle table saves the header field for every object, which consists of four attributes as listed in Table 4.6. Each of these attributes is explained briefly blow.

CTI: Through this attribute, the class of the object can be easily identified. This allows the information about the class to be retrieved efficiently, e.g. the index of its method table which is required upon invoking a method on the object.

Flags: This attribute contains all flags of the class from which the object has been created, with the addition of the GC-specific flags such as those indicating the reachability level of the object.

Name	Width	Description
`CTI`	16-bit	CTI of the class from which the object has been created.
`Flags`	16-bit	Flags of the object.
`Size`	32-bit	Size of the object in bytes.
`Phy_addr`	32-bit	Physical address of the object in the main memory.

Table 4.6: Header field of an object

Size: To facilitate allocating memory for newly created objects and reclaiming memory of dead objects, the size of each object is kept in its header field. If the object corresponds to an array, its size is equal to the product of the array length and element size. Note that the size of a regular object can also be found in the class table, using the CTI above. However, to increase access performance and allow all objects to be handled in one common way, its size is saved in the handle table additionally.

Phy_addr: The value of this attribute provides the physical position of the object in the form of an offset from the start of the heap area to the first field of the object.

To hold the header fields of the immortal objects, a part of the handle table is also generated in advance. The first index of the handle table, i.e. handle 0, is reserved to represent the *null pointer*. Handle 1 is assigned to the static field object. The rest of the handle table stores the header fields of the other pregenerated objects in the following order: the string objects, the **`char`**-arrays and the class objects.

4.2.6 Static Resolution

The static resolution of a bytecode that uses a constant pool index as operand primarily means the replacement of this index with other auxiliary information, e.g. a CTI or RMTI in the context of the AXT format. To keep the offsets of all following bytecodes as intact as possible, only the value of the original operand should be altered, instead of its size. However, the single constant pool property of the AXT format causes a special case to be handled, namely **`ldc`**, which is explained in more detail below.

Additionally, for the purpose of analysis and debugging, not only the operand of a resolved bytecode is replaced, but also its opcode. Currently, there are a total of 51 unassigned bytecodes in the range between 0xCB and 0xFD, which are reserved for future use. 22 among these bytecodes are employed to perform the static resolution and referred to as *quick* bytecodes below. In the following description, the mnemonic of such a bytecode ends with **`_quick`**, to distinguish this bytecode from the original one.

`ldc`, `ldc_w` and `ldc2_w`

These bytecodes load a constant value from the constant pool onto the operand stack. The former two load a 32-bit value, while the latter one a 64-bit value. Unlike **`ldc_w`** and **`ldc2_w`** which have a 16-bit operand, **`ldc`** only has a single-byte operand and is intended to load a value from one of the first 256 constant pool entries. However, due to aggregating multiple constant pools into a single one, a number of constant pool entries must be reordered and therefore placed out of this range. As a result, using an 8-bit index, they cannot be referenced in the aggregated constant pool anymore. To solve this issue, **`ldc`** is treated as **`ldc_w`** during the static resolution, which means an extra byte has to be inserted. This also implies that **`ldc`** does not have a dedicated quick version and must share quick bytecodes with **`ldc_w`**.

Besides numeric constants, **ldc** and **ldc_w** may also reference an constant pool entry of type `CONSTANT_String`. Since string literals have been converted to immortal string objects and removed from the aggregated constant pool, an extra bytecode, namely **ldcs_w_quick**, is adopted to load string objects. This bytecode uses the handle of the corresponding string object as operand. Note that the width of a handle is actually greater than 16 bits (22 bits by default). However, since all immortal string objects are placed successively at the very beginning of the handle table, 16 bits should be enough to reference all of them in most cases. Table 4.7 shows the quick bytecodes used to resolve the three **ldc**-bytecodes, each of which has a 2-byte operand.

Original bytecode	Quick bytecode	Operand description
`ldc/ldc_w`	`ldc_w_quick`	16-bit constant pool index.
	`ldcs_w_quick`	16-bit handle of the corresponding string object.
`ldc2_w`	`ldc2_w_quick`	16-bit constant pool index.

Table 4.7: Quick bytecodes of `ldc`, `ldc_w` and `ldc2_w`

`getstatic` and `putstatic`

The former bytecode pushes the value of a static field of a class onto the operand stack, while the latter performs the inverse operation. Both of them use a 16-bit constant pool index as operand, through which an entry of type `CONSTANT_Fieldref` is referenced.

As described above, the static fields of all classes are combined together to construct a static field object. This means that each static field can be clearly identified by its offset inside this virtual object. However, to access a field properly, its type information is also necessary, which does not exist in the constant pool anymore. Therefore, several additional quick bytecodes are employed to compensate for this lack of information as shown in Table 4.8.

Original bytecode	Quick bytecode	Operand description
`getstatic`	`getstatic_quick`	16-bit offset inside the static field object.
	`getstatic2_quick`	
	`getstatica_quick`	
`putstatic`	`putstatic_quick`	16-bit offset inside the static field object.
	`putstatic2_quick`	

Table 4.8: Quick bytecodes of `getstatic` and `putstatic`

According to the current object layout of the AMDIAR processor, every non-64-bit field is simply assigned 4 bytes on the heap. This means that the size of a field of an arbitrary object, including the static field object, is either 32 or 64 bits regardless of its declaration type[3]. Therefore, **getstatic_quick/putstatic_quick** as well as **getstatic2_quick/putstatic2_quick** would be enough for accessing fields of all types. However, to facilitate tracing references on the operand stack, a dedicated bytecode, namely **getstatica_quick**, is used to get values of reference static fields. As a

[3] Section 4.5.4 discusses the object layout in more detail

value is pushed onto the operand stack by this bytecode, the value is marked as reference. Consequently, the garbage collector can distinguish it from a primitive value during traversing the object graph.

getfield and putfield

These bytecodes are quite similar to **getstatic** and **putstatic**, except that they are adopted to get and set the value of a field of an object rather than a class. Therefore, besides the 16-bit constant pool index, the handle of the target object is also required at runtime. However, for the static resolution, only the constant pool index needs to be replaced with the offset of the field inside the target object, as shown in Table 4.9.

Original bytecode	Quick bytecode	Operand description
getfield	getfield_quick getfield2_quick getfielda_quick	16-bit offset inside the target object.
putfield	putfield_quick putfield2_quick	16-bit offset inside the target object.

Table 4.9: Quick bytecodes of getfield and putfield

invoke-bytecodes

So far, **invokevirtual**, **invokespecial**, **invokestatic** and **invokeinterface** have been implemented in the AMIDAR processor, while **invokedynamic** is currently not supported. The former three **invoke**-bytecodes all have a 16-bit operand that provides the index of a constant pool entry of type CONSTANT_Methodref. In addition to this symbolic reference, **invokeinterface** has another two operands, each of which is given using a single byte. One of them provides the method argument number and the other is a constant 0.

The goal of resolving the **invoke**-bytecodes is that the AMTI of a method must be either directly accessible or able to be calculated efficiently. How **invokevirtual** and **invokeinterface** are resolved has been introduced partially above. An important thing to note is that a method which is invoked via **invokevirtual** or **invokeinterface** has an extra argument, namely the reference to the object on which the method has been called. Only through this reference, the method table that actually contains the invoked method can be found. However, this reference is pushed onto the operand stack before the other arguments of the method. Therefore, the argument number of the invoked method is necessary for locating the reference on the operand stack. As a result, both **invokevirtual_quick** and **invokeinterface_quick** use the method argument number as operand additionally.

The resolution of the remaining two bytecodes, namely **invokespecial** and **invokestatic**, is more straightforward to realize. This is because methods which need to be invoked via these bytecodes are already determined at compile time. Consequently, both bytecodes can use the AMTI of the corresponding method as operand directly. Table 4.10 lists the quick bytecodes employed to resolve the **invoke**-methods.

Original bytecode	Quick bytecode	Operand description
`invokevirtual`	`invokevirtual_quick`	6-bit method argument number. 10-bit RMTI of the method.
`invokespecial/invokestatic`	`invokenonvirtual_quick`	16-bit AMTI of the method.
`invokeinterface`	`invokeinterface_quick`	6-bit method argument number. 10-bit IOLI of the interface. 16-bit method declaration order.

Table 4.10: Quick bytecodes of `invoke`-bytecodes

Object-related bytecodes

The object-related bytecodes include **`new`**, **`checkcast`** and **`instanceof`**. They all refer to a constant pool entry of type `CONSTANT_Class`, using a 16-bit index. In an AXT file, each class is identified by its CTI. Thus, these three bytecodes can be resolved by simply replacing their symbolic reference with the corresponding CTI as illustrated in Table 4.11.

Original bytecode	Quick bytecode	Operand description
`new`	`new_quick`	16-bit CTI of the class from which an object is created.
`checkcast`	`checkcast_quick`	16-bit CTI of the compared class.
`instanceof`	`instanceof_quick`	16-bit CTI of the compared class.

Table 4.11: Quick bytecodes of object-related bytecodes

Array-related bytecodes

There are three array-related bytecodes that need to be resolved statically: **`newarray`**, **`anewarray`** and **`multianewarray`**. **`newarray`** creates an one-dimensional primitive array. Its single-byte operand provides the element type of the array. Since an array is treated as an object, a handle table entry needs to be created for it according to its CTI. Therefore, this bytecode should be resolved in such a way that the CTI of the corresponding array type can be easily determined at runtime. The most straightforward approach to achieve this goal would be replacing the original operand of **`newarray`** with the CTI of the array type directly. However, a single byte can barely hold the entire CTI, because all array types are stored at the end of the class table. For this reason, in stead of the CTI, the IOLAT of the array type is used to resolve **`newarray`**.

`anewarray` and **`multianewarray`** create an one-dimensional reference array and a multi-dimensional array respectively. Unlike **`newarray`**, both of them have a 16-bit symbolic reference to the constant pool, which identifies the array type. For the static resolution, the symbolic reference of both bytecodes is simply replaced with the CTI of the array type. Additionally, **`multianewarray`** also has an 8-bit operand that provides the number of the dimensions, which remains unchanged. Table 4.12 summarizes the quick bytecodes of the three array-related bytecodes.

Original bytecode	Quick bytecode	Operand description
`newarray`	`newarray_quick`	8-bit IOLAT of the array type.
`anewarray`	`anewarray_quick`	16-bit CTI of the array type.
`multianewarray`	`multianewarray_quick`	16-bit CTI of the array type. 8-bit dimension number.

Table 4.12: Quick bytecodes of array-related bytecodes

4.2.7 Evaluation

Due to the static class linking and resolution, the AXT format has several limitations on applications that need to run on the AMIDAR processor, including:

- The number of classes (inclusive interfaces and array types) that may be declared in an application is limited to 2^{16}.
- The number of methods that may be declared in an application is limited to 2^{16}.
- The number of instance methods that may be declared in a class is limited to 2^{10}.
- The number of static methods that may be declared in a class is limited to 2^{16}.
- The number of fields that may be declared in an application is limited to 2^{16}.
- The number of instance fields that may be declared in a class is limited to 2^{16}.
- The number of static fields that may be declared in a class is limited to 2^{16}.

To evaluate the influence of these limitations on the application scale, a web-based administration software used in the room automation station from *Sauter AG* [90] was chosen as the evaluation benchmark and analyzed in detail. The AXT file generated from the benchmark software includes a total of 828 classes. The numbers of the instance/static methods and the instance/static fields defined in all these classes are shown in Table 4.13. Additionally, this table also summarizes the maximum numbers of the instance/static methods and the instance/static fields defined in a single class of the benchmark software. As the table illustrates, every measured value is far below the corresponding limit, which indicates that the AXT format should be able to be applied to a broad variety of applications.

	Instance methods	Static methods	Instance fields	Static fields
Total number$_{\text{AXT}}$	8365	994	1636	2549
Max. number$_{\text{Class}}$	187	96	72	477

Table 4.13: Measurement results for analyzing the AXT limitations

One of the key design goals of the AXT format is to make an AXT file as compact as possible. For the purpose of the compactness evaluation, a JAR file and a DEX file were generated from all classes included in the AXT file of the benchmark software. The sizes of these files are shown together with the

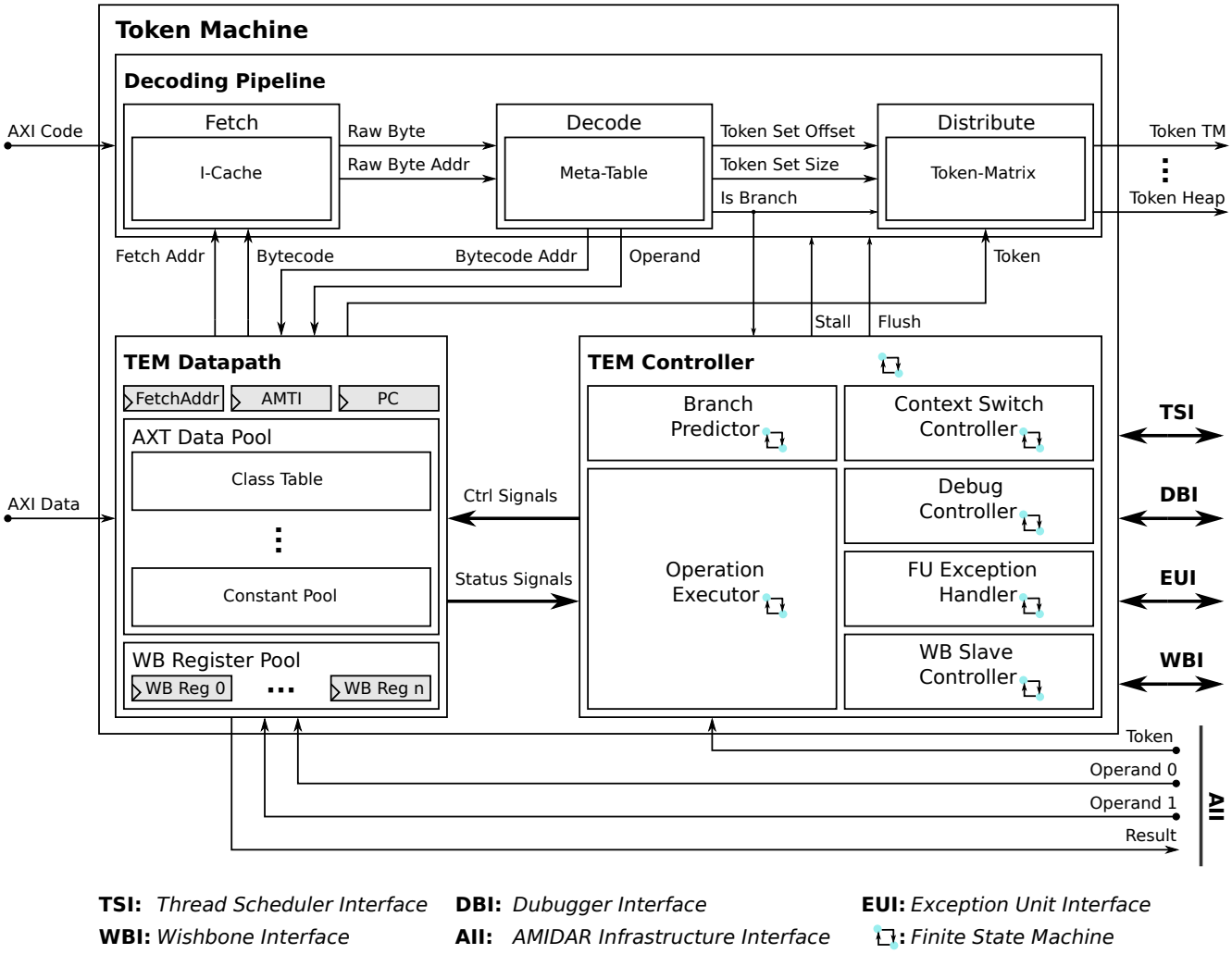
total size of the original class files in Table 4.14. Clearly, AXT is the most compact one among all these four formats. In comparison with the class format, the size of the code memory required for storing the benchmark software can be reduced by 71.42% through using the AXT format.

AXT	JAR	DEX	Class	Δ% (Class ⟷ AXT)
1.6 MB	1.8 MB	2.1 MB	5.6 MB	71.42

Table 4.14: Compactness comparison among different formats

4.3 Token Machine

The token machine is the centerpiece of the AMIDAR processor, which has two major tasks. On the one hand, it drives the whole system by generating and distributing tokens to different FUs. On the other hand, the token machine itself is also an FU and needs to execute tokens delivered to it. Therefore, the token machine can be logically considered as a combination of a decoding pipeline and a token execution module (TEM) that is composed of a controller and a datapath, as shown in Figure 4.3.



TSI: *Thread Scheduler Interface* **DBI:** *Dubugger Interface* **EUI:** *Exception Unit Interface*
WBI: *Wishbone Interface* **AII:** *AMIDAR Infrastructure Interface* **:** *Finite State Machine*

Figure 4.3: Token machine overview

The 3-stage decoding pipeline works autonomously as long as bytecodes are executed in sequence. The TEM interferes with it only upon the occurrence of one of the following cases:

- Execution of a branch bytecode like **`goto`** or **`invokestatic`**.
- Thread context switching.
- Exception handling.
- Hardware-level debugging.

In the following, we introduce both logical parts and their interactions in detail.

4.3.1 Decoding Pipeline

The decoding pipeline is composed of three stages, namely *fetch*, *decode* and *distribute*.

Fetch Stage

Figure 4.4 demonstrates the internal structure of the fetch stage. It consists of a fetcher, an instruction cache (i-cache), a prefetch queue, a word-to-byte converter as well as an output buffer. All these components have been designed and implemented in such a way that the fetch stage can provide a byte every clock cycle to the next pipeline stage, when bytecodes are executed sequentially.

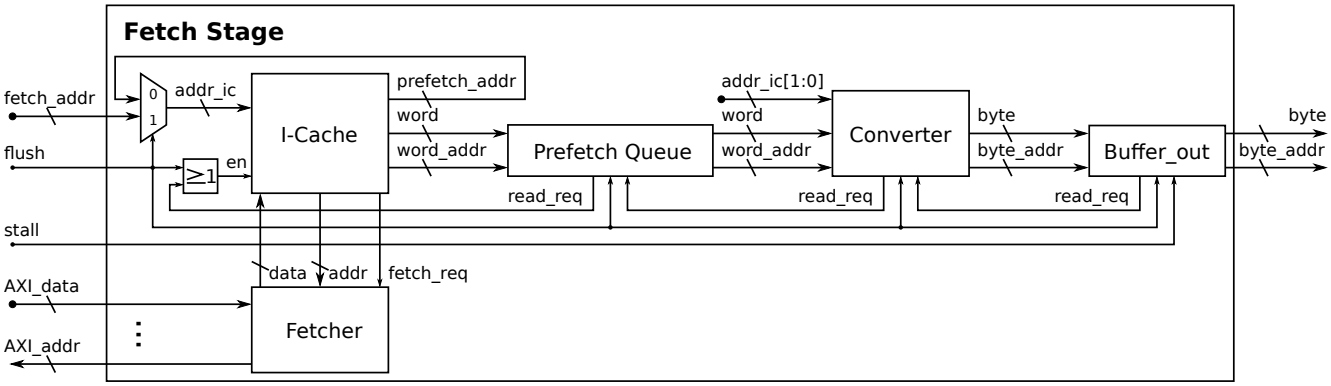


Figure 4.4: Fetch stage

The fetcher facilitates accessing to the DRAM-based main memory of the AMIDAR processor. Through an AXI master interface, it is connected to a memory controller generated by the *memory interface generator* (MIG) from Xilinx. For the Artix-7 FPGA which is used as the platform of the AMIDAR processor [75], the minimum data width of the memory controller is fixed to 32 bits [124]. On an i-cache miss, the fetcher fills a new cache line via a burst transfer.

The i-cache is a 4-way set-associative cache using the Pseudo-LRU replacement policy. Like the fetcher, its data width is 32 bits as well. Consequently, given the address of a byte, the i-cache simply returns the entire word containing the byte. To extract the actually required byte later, its offset inside the result word ($addr\_ic[1:0]$) needs to be forwarded to the converter in addition.

When the $flush$-signal is asserted, the i-cache yields the word addressed by $fetch\_addr$. $fetch\_addr$ is byte-aligned to allow every bytecode to be uniquely addressed, which is critical for executing branch bytecodes. Depending on both lowest significant bits of $fetch\_addr$, the result word can

contain up to 3 redundant bytes, which will be removed by the converter. If the required byte has not been fetched into the cache yet (i.e. a cache miss occurs), a new cache line is loaded by the fetcher, using $fetch\_addr$ as the start address. If $fetch\_addr[1:0]$ is equal to 11_2, this cache line includes a total of $4n_{CL}-3$ valid bytes, where n_{CL} represents the cache line size.

Besides the result word, the i-cache also outputs its address ($word\_addr$) and the address of the word following behind the result word ($prefetch\_addr$). Unlike $fetch\_addr$, both addresses are always word-aligned. After the *flush*-signal becomes inactive, the i-cache can fill the prefetch queue autonomously by returning the word addressed by $prefetch\_addr$ upon a read request from the prefetch queue. Note that a cache line fetched at address $prefetch\_addr$ does not contain invalid bytes.

The FIFO-based prefetch queue aims to minimize the impact of stalls caused by i-cache misses. It can store all words of a single cache line and their addresses. As long as it is not full, it reads a new word from the i-cache. After a pipeline flush, the first cache line read from the i-cache can hold $4n_{CL}-3$ valid bytes in the worst case, as discussed above. Assume that the decode stage consumes a byte every clock cycle. To ensure that the decoding process is not interrupted by an i-cache miss, the prefetch queue may not become empty until the missing cache line is fetched. This condition can be formally described as follows:

$$4n_{CL}-3 > t_{DRAM}$$

where t_{DRAM} corresponds to the average DRAM access delay in clock cycles. According to the measurements of multiple benchmarks, t_{DRAM} is about 11 cycles on the *Nexys Video* FPGA board [75] when the AMIDAR processor operates at 100 MHz. As a result, the current cache line size of the i-cache is set to 4.

The word-to-byte converter transforms an incoming word to four bytes. For each of them, a byte-aligned address is generated at the same time. Also, it removes invalid bytes based on the given offset. The whole conversion takes as many clock cycles as the number of the valid bytes and the first valid byte becomes available one cycle after the arrival of the word.

Like the prefetch queue, the output buffer is also based on a FIFO architecture. It can store up to four bytes and their physical addresses. Once it has a free entry, it reads a new byte from the converter.

After a system reset, the TEM of the token machine asserts the $stall$-signal until the processor has been initialized completely (e.g. writing 0 across the main memory). The default value of $prefetch\_addr$ is set to the begin address of the bootloader method. As a result, the boot loading process is started automatically after a system reset. Another important thing to note is that the TEM of the token machine can insert additional bytecodes between the fetch and decode stages to alter the original bytecode stream. This feature is necessary for handling exceptions.

Decode Stage

This stage can be further partitioned into two parts: the decoding and debugging logics. The former logic is active by default, whereas the latter one is activated only if an application is started in the debugging mode.

The decoding logic is made up of a read-only meta-table, a simple controller as well as several small FIFO-based output buffers. The first byte provided by the fetch stage after a pipeline flush or system reset is always the opcode of a bytecode. This ensures that the decoding logic can distinguish the following

bytecodes from each other properly. Between two flushes, the decoding logic loads bytecodes from the fetch stage continuously as long as none of its output buffers is full.

Once a new bytecode arrives, the decoding logic first reads out its meta-information from the meta-table, using the opcode of the bytecode as index. The BRAM-based meta-table is initialized with the binary output file of the ADLA compiler and yields the required information in a single clock cycle. In the following clock cycle, the offset and size of the token set of the bytecode are written into the corresponding output buffers and are therefore accessible to the distribution stage.

If the bytecode has operands, the decoding logic redirects them into its operand output buffer towards the TEM of the token machine, one per clock cycle. Concurrently, the distribution stage can already begin distributing tokens to corresponding FUs.

To facilitate executing branch bytecodes, the jump-flag of the bytecode is connected to both distribution stage and TEM of the token machine, as shown in Figure 4.3. For the latter, the physical address of the bytecode is also necessary so that it can determine the new fetch address according to the branch offset. If the jump-flag is set, the decode stage is suspended immediately, while the distribution stage is suspended after all tokens of the current bytecode have been delivered. Both of them can be reactivated via a pipeline flush.

If an application is started in the debugging mode of the AMIDAR processor, the debugging logic is activated and tracks the following events:

- Occurrence of bytecode **`breakpoint`**.
- Occurrence of bytecode **`athrow`**.
- Finish of a step, if the AMIDAR processor is running in the stepping mode.

To detect the former two events, the debugging logic simply compares the opcode of every incoming bytecode with those of **`breakpoint`** and **`athrow`**. To detect the latter one, it compares the address of each bytecode with the step start and end addresses provided by the hardware debugger. A step is considered to be finished, if the address of the current bytecode is outside the step range.

Once one of the three events is detected, the debugging logic sets an event-specific flag, which causes that both decoding pipeline and TEM of the token machine are halted. Then, the hardware debugger takes over the control of the AMIDAR processor. Through the hardware debugger, a programmer can check the status of every object in an interactive manner. After the hardware debugger has given the control back, the TEM converts the current PC to a fetch address which results from adding the code section offset (`Code_section_off`, see Section 4.2.2), the offset of the current method inside the code section (`Code_off`, see Section 4.2.3) as well as the PC. With the calculated address, the FU-logic flushes the decoding pipeline to reactivate both decode and distribute stages.

Distribute Stage

The major part of this stage is a BRAM-based token-matrix, which is initialized using the binary output of the ADLA compiler, just like the meta-table described above. The token set of a bytecode can be located via its offset provided by the decode stage. Each row of the token set is read out in a single clock cycle. In the following clock cycle, the valid tokens held in this row are pushed into the FIFO-based

token queues of the corresponding FUs with the current tag. At the same time, the tag is incremented if the INC-flag of this row is set.

The distribute stage works in a pipeline fashion, i.e. it takes a total of n cycles to distribute a token set of size n. After the last row of the current token set has been extracted from the token-matrix, the offset of the next token set is loaded from the decode stage. This means that the distribute stage can generate tokens every clock cycle as long as the output buffer of the decode stage is not empty and no token queue of any FU becomes full.

If the distribute stage is halted by the asserted *stall*-signal, the TEM of the token machine may insert additional token rows between it and individual FUs. This feature is required during thread context switching. To distinguish inserted tokens from the original ones, the TEM increments the tag after sending an additional token row.

4.3.2 Datapath of the Token Execution Module

The centerpiece of the datapath is the AXT data pool containing multiple read-only tables such as the class and method tables. Each of these tables is associated with the counterpart of an AXT file and provides corresponding information to the TEM. Since the constant pool of an AXT file only stores numeric constants, it has been simply realized as a number table.

Unlike the meta-table and token-matrix built into the decoding pipeline, all these tables have been implemented using a direct mapped cache structure, to reduce the usage of on-chip memory. To distinguish them from the tables saved in the main memory, they are referred to as table caches below.

Every table cache includes a total of 256 sets and is addressed by a 16-bit index. The cache line size of a table cache is determined by the attribute size and number of a single entry. For example, the width of the exception table cache is 8 bytes due to the four 16-bit attributes of an exception handler. An important thing to note is that all methods, regardless of static or instance ones, must be accessed through the method table cache using their AMTIs.

All table caches share one common fetcher module. This module can load a single cache line for an arbitrary table cache through a burst transfer, according to a given table index and cache line size. Before the fetcher starts the transfer, it first needs to convert the table index to a physical address of the main memory. For this purpose, the base address of the table is needed. Although this address can be found in the AXT header stored in the main memory, it is written to a dedicated WB register of the token machine during booting, allowing for a quick access to it.

Besides the AXT data pool, the datapath contains a number of WB registers. They can be accessed at the software level and serve as the communication medium between the hardware and software. Some of them are just employed to keep constant values, e.g. the base addresses of all tables, to assist the TEM with executing some of its operations. The others expose the internal status of the token machine to programmers, e.g. the number of pipeline stalls. In addition, the datapath also includes a number of internal registers to buffer intermediate values produced by the TEM at runtime such as the AMTI and PC of the current method.

4.3.3 Execution of Tokens

The key task of the TEM is to execute tokens delivered to it. Like every other FU of the AMIDAR processor, it has an standard AMIDAR infrastructure interface (AII) as well. Through this interface, it

receives tokens as well as their operands, and then sends their results to the destination FUs. Appendix B.1 lists all operations supported by the TEM of the token machine.

In the following, we explain how **invokevirtual_quick** is performed, especially on the token machine's side, to illustrate how the TEM executes tokens and how it interacts with the decoding pipeline. Listing 8 shows the token set of **invokevirtual_quick**.

Listing 8: Token set of invokevirtual_quick

```
 0: invokevirtual_quick
 1: #BYTECODE_OFFSET(2)
 2: #JUMP_BYTECODE
 3: {
 4:   T(tokenmachine,
 5:     SENDCONSTANT($BYTECODE(1) & $BYTECODE(2)), tokenmachine.0)++;
 6:   T(tokenmachine, LOAD_ARG_RMTI, framestack.0),
 7:   T(framestack,   PEEK,          heapmanager.0),
 8:   T(heapmanager,  GET_CTI,       tokenmachine.0)++;
 9:   T(tokenmachine, INVOKE,        framestack.0),
10:   T(framestack,   INVOKE)
11: }
```

#BYTECODE_OFFSET(2) and #JUMP_BYTECODE indicate that **invokevirtual_quick** has 2 operand bytes and is a branch bytecode. Based on the meta-information of **invokevirtual_quick**, the decode stage first writes both operand bytes into the operand buffer and then suspends itself due to the asserted jump flag. For the same reason, the distribute stage also suspends itself after all tokens of **invokevirtual_quick** haven been distributed.

Upon the arrival of the first token (line 4 and 5), the TEM reads out the first two bytes from the operand buffer of the decode stage and concatenates them as a single 16-bit value. Then, this value is sent to input port 0 of the TEM itself and serves as the operand of the next token. As explained in Section 4.2.6, the high 6 bits of this value correspond to the argument number and the low 10 bits the RMTI of the invoked method.

To execute the second token delivered to it (line 6), the TEM first loads the argument number and RMTI of the invoked method from input port 0 into two internal registers. However, both values are still not enough for determining the AMTI of the method. This is because the TEM does not know on which object the method has been called and therefore is not able to locate the proper method table. To solve this issue, the TEM sends the argument count as the result of the second token to the frame stack.

Exploiting the argument count provided by the token machine, the frame stack can address the handle of the object on which the method has been called in the operand stack. It simply transmits this value to the heap manager without removing it from the operand stack (line 7). According to the given object handle, the heap manager reads out the corresponding CTI from the handle table and returns it back to the token machine (line 8).

Now, the TEM has all necessary information to carry out its last token (line 9). First, it obtains the index of the required method table (IMT, see Section 4.2.3) from the class table cache, using the CTI

returned by the heap manager. After that, the AMTI is calculated by adding the IMT to the buffered RMTI. In the following step, the AMTI of the current method (i.e. the caller) and its PC, along with the argument and local variable number of the invoked method (i.e. the callee) are packed into a single 64-bit result and sent to the frame stack. Finally, the TEM updates its AMTI and PC registers and then flushes the decoding pipeline with the start address of the bytecode stream of the callee in the main memory. How the frame stack generates a new frame for the callee (line 10) is a topic for Section 4.4 below.

Branch Prediction

Inside the AMIDAR processor, a conditional branch bytecode is executed as follows. The frame stack first pops one or two values from the top of the operand stack and sends it or them to the integer ALU. If a single value is popped, it is compared with 0; otherwise, both values are compared with each other. The result of the comparison is transmitted to the TEM of the token machine. If the comparison succeeds (i.e. the result is equal to 1), the TEM performs a jump. To do this, it first calculates the target address of the conditional branch by adding the branch offset given as the operand of the bytecode to the physical address of the bytecode. Then, it flushes the decoding pipeline with this address. In the meantime, the PC is also updated accordingly.

As described above, the TEM must perform a comparison before it can determine how a conditional branch bytecode should be executed, i.e. whether the conditional branch should be taken or not. Consequently, the decoding pipeline must be stalled for a number of clock cycles until the comparison result is available. It would cost even more cycles if the branch needs to be taken, but the corresponding bytecodes have not been loaded into the i-cache yet.

To reduce stall cycles caused by executing conditional branch bytecodes, a small branch predictor has been built into the TEM. It simply presumes that backward branches will always be taken. The reason is that loops are typically realized based on backward branches at the bytecode level[4], which are therefore taken more often than not taken.

A backward branch can be identified by a negative branch offset. Once such a branch offset is detected when executing a conditional branch bytecode, the TEM starts calculating the target address of the branch immediately without waiting for the comparison result. Then, it solely flushes the fetch stage with the generated address, which causes that a new cache line is fetched into the i-cache. This cache line includes the first bytecodes of the branch. If the prediction is correct, the TEM only needs to clear the $stall$-signals asserted for both decode and distribute stages. Otherwise, it must reflush the whole decoding pipeline with the address of the bytecode following the conditional branch bytecode.

4.3.4 Exception Handling

In Java, exceptions are classified into two groups: checked and unchecked ones. A checked exception must be explicitly thrown using the keyword **`throw`**. An unchecked exception can be thrown in the same way, which is however unnecessary, since throwing unchecked exceptions is supposed to be the task of the underlying runtime system.

[4] This is the case if Oracle JDK or OpenJDK is used.

During the implementation of the AMIDAR processor, the exception handling scheme of Java was slightly extended to make it fit into the AMIDAR model. In the extended scheme, exceptions are reclassified according to their sources into software and hardware exceptions. A software exception is thrown via the keyword **throw** at the software level, while a hardware exception is thrown by an FU. Based on this classification, an unchecked exception like **NullPointerException** can either be a software or a hardware exception, depending on how it is thrown. Below, we explain why this extension is required and how it is implemented in the AMIDAR processor.

Handling Software Exceptions

The keyword **throw** is translated to bytecode **athrow** at compile time. The token set of **athrow** is demonstrated in Listing 9.

Listing 9: Token set of athrow

```
 0: athrow
 1: #JUMP_BYTECODE
 2: {
 3:   T(framestack,   DUP),
 4:   T(framestack,   POP32,   heapmanager.0)++;
 5:   T(heapmanager,  GET_CTI, tokenmachine.0)++,
 6:   T(framestack,   POP32,   tokenmachine.1)++;
 7:   T(framestack,   CLEARFRAME),
 8:   T(tokenmachine, THROW,   framestack.0)++;
 9:   T(framestack,   PUSH32)
10: }
```

Once an **athrow** enters the decode stage, it halts the decoding pipeline due to its asserted jump flag. As a result, the AMTI and PC registers of the TEM become frozen, which is critical for the exception handling process, as explained below.

To expose detailed information about the exception to the token machine, the frame stack duplicates the handle of the exception object on the operand stack (line 3). One of both handles is sent to the heap manager to get the CTI of the corresponding exception class (line 4 and 5), while the other is delivered to the token machine (line 6). Then, the frame stack clears the operand stack of the current method (line 7).

As soon as both operands (i.e. the CTI of the exception class and the exception object handle) have been received, the TEM of the token machine starts handling the exception (line 8). First, the exception object handle is returned back to the frame stack so that it can be pushed onto the cleared operand stack again (line 9). According to the Java Virtual Machine specification [50], the exception object handle must be the only value on the operand stack after executing **athrow**.

In the next step, the TEM reads out the exception table index and size from the method table cache, using the AMTI. Then, it searches the cached exception table for a matching handler in a top-down fashion, exploiting an FSM that is intended to realize bytecode **instanceof**. This FSM can determine if a given class A is derived from another class B by recursively comparing the CTI of B with that of every

superclass of *A*, whether direct or indirect, until **`java.lang.Object`** is reached. An exception table entry matches only if both of the following conditions are met:

- The current PC is within the range of the entry.
- The CTI returned by the heap manager is equal to that of the catch type specified by the entry or belongs to a subclass of the catch type.

When the first match is found, the TEM replaces the current PC with the handler PC specified by the found entry. After that, it converts the new PC to a fetch address and flushes the decoding pipeline with this address.

If no handler can be found, the TEM needs to search the exception table of the caller of the current method further. For this purpose, it inserts two additional bytecodes into the original bytecode stream: an **`areturn`** behind the current **`athrow`** and an **`athrow`** behind the invoke bytecode of the caller. Consequently, the exception is re-thrown in the caller and the TEM will repeat the exception handling process described above.

Handling Hardware Exceptions

The challenge of handling a hardware exception is the synchronization between the token machine and the FU throwing the exception. There can be a delay of multiple clock cycles from the time point at which the causing bytecode is decoded to the time point at which the exception is actually thrown. During this time, both AMTI and PC can vary. This has the consequence that the token machine is no longer in the context in which the exception was caused, when the exception is thrown.

To solve this issue, a simple tracing module called *exception unit* was developed. It is not an FU of the AMIDAR processor and is only intended to keep the latest context information of the token machine. Its key component is a BRAM-based context table with 2^{W_T} entries, where W_T represents the tag width in bits. Each of its entries can hold a 32-bit AMTI-PC pair. As Figure 4.5 illustrates, the exception unit is connected to both token machine and FUs that can throw exceptions. All signals on the left side belong to the exception unit interface (EUI) of the token machine.

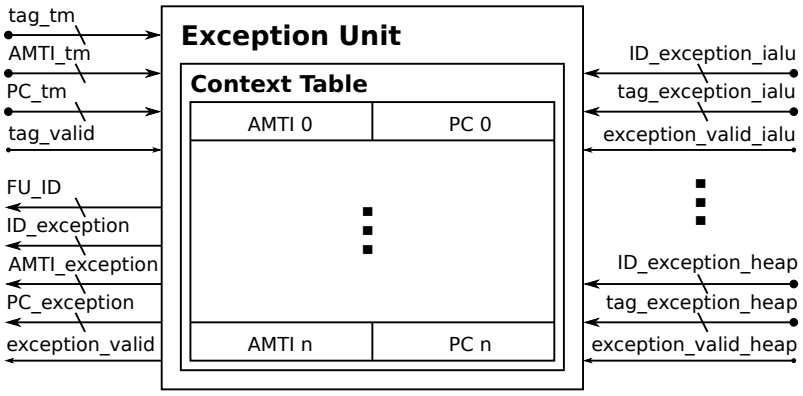


Figure 4.5: Exception unit

Every time the distribute stage increments the tag, the TEM of the token machine sends the current AMTI and PC with the new tag to the exception unit. These values are saved into a single entry of the context table, which is indexed by the tag.

If some FU throws an exception at runtime, it interrupts its current operation and asserts its *exception_valid*-signal to notify the exception unit. Simultaneously, the FU also outputs the tag of the token causing the exception as well as the ID of the exception. Using the tag, the exception unit first reads out the corresponding AMTI and PC. Then, it sends both values together with the FU and exception IDs to the token machine.

Once the *exception_valid*-signal of the exception unit becomes active, the TEM of the token machine halts the decoding pipeline immediately. After that, the values provided by the exception unit are written into four WB registers used to assist with handling hardware exceptions. Then, the PC register is reset, while the AMTI register is set to the AMTI of a static method called **hardwareExceptionHandler** defined in class **de.amidar.AmidarSystem**, as shown in Listing 10. At last, the decoding pipeline is flushed with the physical address of this method.

Listing 10: Hardware exception handler

```
 0: static void hardwareExceptionHandler() throws RuntimeException {
 1:   Tokenmachine tm = Tokenmachine.Instance();
 2:   // output exception information
 3:   System.err.println ("Hardware exception");
 4:   System.err.println ("FU:   " + tm.EXCEPTION_FU);
 5:   System.err.println ("ID:   " + tm.EXCEPTION_ID);
 6:   System.err.println ("AMTI: " + tm.EXCEPTION_AMTI);
 7:   System.err.println ("PC:   " + tm.EXCEPTION_PC);
 8:   // throw an unchecked exception
 9:   throw new RuntimeException ("Hardware exception");
10: }
```

Class **de.amidar.Tokenmachine** is the abstract representation of the token machine at the software level. It contains a set of fields that can be mapped to the WB registers of the token machine. Only a single instance of this class can be created via the **Instance**-method (line 1). Through this instance, the information of a hardware exception is accessible to programmers (line 4-7). The **main**-method of a Java application is invoked by the bootloader method inside a **try-catch** block. Therefore, the unchecked exception thrown at the end of the **hardwareExceptionHandler**-method will be handled there if it is not caught inside the **main**-method.

4.3.5 FU Interfaces

Interface to Hardware Debugger

As described in Section 4.3.1, the decoding pipeline tracks three events, if an application is started in the debugging mode of the AMIDAR processor. Once an event occurs, the decoding pipeline is halted and a flag signal indicating the type of the event is asserted. As response to this event, the hardware debugger sets a request signal. This causes that the TEM of the token machine enters a waiting state and stays in the state as long as the request signal is active. During this time, the hardware debugger plays the role of the token machine.

Under control of a programmer, the hardware debugger can deliver tokens to the heap manager to access the object cache and the entire main memory from the code section to the heap. This allows the programmer to obtain detailed ongoing information about the application and thus eases locating bugs. If the programmer starts stepping the Java code, the hardware debugger provides a start and end PC for every step, which are converted to the physical addresses of the corresponding bytecodes by the TEM on the fly. To allow a step to be performed, the hardware debugger cancels its request temporarily.

An important thing to note about **breakpoint** is that this bytecode is not generated by the Java compiler but added by the hardware debugger. According to the position of a breakpoint set by the programmer, the hardware debugger overwrites an original bytecode with **breakpoint** in the main memory directly through the heap manager. The replaced bytecode is returned to the software part of the AMIDAR debugging framework, which is referred to as the software debugger below. The software debugger runs on an independent host computer and communicates with the hardware debugger over a JTAG connection. If an application that has been halted by a breakpoint needs to proceed, the original bytecode is provided by the software debugger to the decoding pipeline over the hardware debugger. If a breakpoint is removed, the replaced bytecode is written back to its original position in the main memory.

Interface to Thread Scheduler

A thread context switch can be caused by either the token machine itself or the thread scheduler. The former case occurs upon the invocation of a thread blocking method like **sleep** or **join** of class **Thread**. This has the consequence that the TEM of the token machine halts the decoding pipeline immediately. At the same time, it asserts a signal called *CS_waiting* to request a context switch from the scheduler. The latter case occurs, if the time-slice of the current thread has expired or an interrupt request has been issued from some peripheral device.

In both cases, the scheduler asserts a signal called *CS_request* to notice the token machine about the triggered context switch. However, in the former case, this signal serves as an acknowledgement rather than a request. In the latter case, the TEM suspends the decoding pipeline after it has received the request from the scheduler.

After the assertion of *CS_request*, the TEM waits until the token queues of all FUs become empty. Then, it sends the AMTI and PC of the current thread to the scheduler which returns the AMTI, PC and ID of the next thread back. Based on the new AMTI and PC, the TEM calculates a fetch address for the next thread.

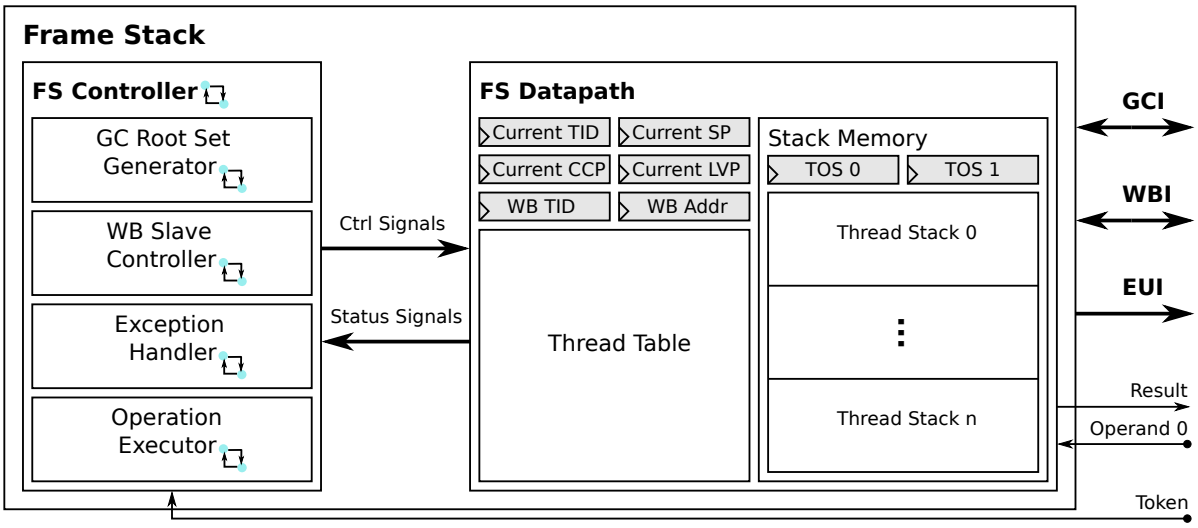
Before flushing the decoding pipeline, the TEM first inserts an additional token row into the distribute stage, which only contains two tokens: `T(tokenmachine, THREADSWITCH, framestack.0)` and `T(framestack, THREADSWITCH)`. The first makes the TEM transmit the next thread ID to the frame stack, while the second results in a stack switch inside the frame stack, which is explained in the following section. Finally, the TEM flushes the decoding pipeline to start the execution of the next thread.

4.4 Frame Stack

The frame stack is a straightforward implementation of the Java stack described in Section 2.3.1. It includes a private Java stack for every thread alive and allows a context switch to be performed in a total of 6 clock cycles. Exploiting a few pointer registers, it can generate and eliminate stack frames efficiently.

Also, it provides a number of operations to support speedy access to the operand stack and local variable array of the current frame. Furthermore, it supplies the garbage collector with the major part of the root set, which contains the object handles that are stored on the Java stacks of all threads alive.

Figure 4.6 provides a brief overview on the structure of the frame stack. In the following, we first describe the frame stack's datapath components and then introduce how its major functions are realized by using these components.



GCI: *Garbage Collector Interface* **WBI:** *Wishbone Interface* **EUI:** *Exception Unit Interface*

Figure 4.6: Frame stack overview

4.4.1 Datapath Components

The datapath of the frame stack is composed of a stack memory, a thread table as well as several pointer and WB registers, which are explained separately below.

Stack Memory

The stack memory is the centerpiece of the frame stack, which has been implemented as a random access memory whose address space starts at 0. It is solely based on BRAMs without using off-chip memory. This design decision was made for several reasons. First, the AMIDAR processor is intended to be used in the field of embedded systems. According to the measurements of real-world programs from this field, which are introduced in Section 5.1, a stack size of 8 kilobytes is large enough to meet the requirements of all these programs. Even the SPEC JVM98 benchmarks [99] do not need a stack size larger than 16 kilobytes, as illustrated in Appendix A. Second, the number of BRAMs available in modern FPGAs increases rapidly. For example, the Artix-7 FPGA [75] which is used as the platform of the AMIDAR processor provides over 1.6 megabytes BRAM-based on-chip memory. This means that less than 0.5% of the BRAMs will be utilized by a Java stack of size 8 kilobytes. Third, the frame stack is currently the most loaded FU, because it must assist with executing almost all Java bytecodes. Thus, we favor time over space at this point to increase the performance of the entire system. Also, this design allows for a quick thread context switch and eases generating the root set for the garbage collector, which are discussed in more detail later.

The width of the stack memory is 34 bits. Both highest significant bits provide the type information about an entry, while the remaining 32 bits hold the data value stored in this entry. A total of four entry types are defined, as shown in Table 4.15.

Type	Value	Description
`Empty`	00_2	The entry is unused.
`Value`	01_2	The entry contains a value of some primitive type.
`Handle`	10_2	The entry stores an object handle.
`CC`	11_2	The entry holds caller context data.

Table 4.15: Entry types of the stack memory

The stack memory is partitioned into multiple Java stacks of same size. Every Java stack includes 2^n contiguous entries. The lowest n address bits of an entry represent its offset inside the Java stack containing it, while the remaining address bits represent the Java stack's number. As a thread is created, it is assigned a dedicated Java stack whose number is equal to its ID. Therefore, the absolute address of a stack entry of some thread can be determined by concatenating the thread's ID and the entry's offset.

Each time a thread invokes a method, a stack frame is pushed onto the Java stack of the thread. In the AMIDAR processor, a stack frame is divided into three consecutive areas as follows: an operand stack, a frame data section and a local variable array. This structure allows for overlapping the stack frames of both caller and callee, eliminating the need for copying parameters from the caller's operand stack to the callee's local variable array. How a stack frame is constructed upon a method invocation is explained in more detail in Section 4.4.2. Since the frame data section solely stores the context information about the caller, it is referred to as the caller context section below.

At a time, only one thread can run on the AMIDAR processor. Its current frame is said to be the current frame of the frame stack and its ID is saved in the *current TID* register illustrated in Figure 4.6. Three pointer registers are employed to keep the context information about the current frame, namely *stack pointer* (SP), *caller context pointer* (CCP) and *local variables pointer* (LVP). The meaning of each of these pointers is described in Table 4.16.

Pointer	Description
`SP`	Position of the first unused entry on the operand stack.
`CCP`	Position of the first entry in the caller context section.
`LVP`	Position of the first entry in the local variable array.

Table 4.16: Pointer registers of the current frame

Besides the pointer registers, there is another register called *max pointer* (MP), which is solely used to track the maximum stack depth. Note that the values stored in all these four registers are offsets from the base address of the current Java stack, rather than absolute addresses. Additionally, there is a single bit flag register that is used to indicate whether an overflow has occurred on the current Java stack.

Since the first two values on the top of the operand stack are used frequently, they are buffered into separate registers, namely *TOS 0* and *TOS 1* shown in Figure 4.6. This allows both values to be accessed in a single clock cycle.

Thread Table

The BRAM-based thread table saves the SP, CCP, LVP, MP as well as the overflow flag of the Java stack of every inactive thread. It inherits the dual port property from BRAM, which allows values of two different threads to be accessed at the same time. Each value of a thread can be addressed using the thread's ID (i.e. the row number of the value) and the value's ID (i.e. the column number of the value).

WB Registers

To ease the interaction between hardware and software, the frame stack also implements the WB interface, like the token machine. Class **de.amidar.FrameStack** is the abstract representation of the frame stack at the software level. Listing 11 lists several important fields defined in this class, through which programmers can access the thread table and stack memory at runtime.

Listing 11: Fields defined in class de.amidar.FrameStack

```
 0: // access to the thread table
 1: public int threadSelect;
 2: public int localsPointer;
 3: public int callercontextPointer;
 4: public int stackPointer;
 5: public int maxPointer;
 6: public int overflow;
 7: // access to the stack memory
 8: public int stackAddressSelect;
 9: public int stackData; // 32-bit data value
10: public int stackMeta; //  2-bit type value
```

Note that only **threadSelect** (line 1) and **stackAddressSelect** (line 8) are actually mapped to WB registers of the frame stack, namely *WB TID* and *WB Addr* shown in Figure 4.6. The other fields are mapped to the entries in the thread table and stack memory directly.

For example, to access a value of some thread held in the thread table, the thread's ID must be written into the WB TID register by assigning it to **threadSelect**. After that, upon any access to one of the five fields following **threadSelect** (line 2-6), the WB slave controller first determines the ID of the corresponding value based on the offset of the field. Then, it combines this ID with the thread's ID to locate the position of the value. To access a Java stack entry of the thread, the offset of the entry must be assigned to **stackAddressSelect** in addition. In Section 4.6.2, we explain how to initialize the Java stack of a newly created thread by using the WB interface of the frame stack.

4.4.2 Execution of Tokens

The frame stack provides a number of operations described in Appendix B.2. All these operations have at most one operand, except `INVOKE` which requires four. However, since the width of each of these operands is not greater than 16 bits, they are sent to the frame stack as a single 64-bit data packet, as mentioned in 4.3.3. For this reason, the frame stack only has one data input port.

Based on their functions, the operations provided by the frame stack fall into four broad groups:

- Load and store (e.g. `LOAD32_0`, `STORE64_1`).
- Operand stack management (e.g. `PUSH64`, `DUP`).
- Stack frame management (e.g. `INVOKE`, `RETURN32`).
- Thread management (e.g. `THREADSWITCH`).

All operations in both former groups are performed on the current frame. By exploiting the random access support provided by the stack memory, they are straightforward to implement. Therefore, the following discussion is solely focused on the latter two operation groups.

Stack Frame Management

A total of five operations have been implemented to assist with managing stack frames, namely `INVOKE`, `RETURN`, `RETURN32`, `RETURN64` and `CLEARFRAME`. The first operation creates a new stack frame on the current Java stack and the second discards the current frame. `RETURN32` and `RETURN64` discard the current frame as well, but in addition, return a 32-bit and 64-bit result respectively. The last one resets the current frame to its initial state, i.e. clears its operand stack completely. Below, we explain the structure of a stack frame and its lifetime in detail, based on a simple example.

Assume that a method has two 32-bit arguments and returns a 32-bit result. Figure 4.7 illustrates the states of the current Java stack from before the invocation to after the completion of the method. As the figure shows, the stack frames of both caller and callee overlap partially. The portion of the caller's operand stack that contains the parameters of the callee becomes the base of the callee's local variable array. This avoids the need for copying parameters between these two stack frames. Consequently, the creation of the stack frame of the callee simply requires resetting the three pointer registers.

`INVOKE` has four operands, namely the AMTI and PC of the caller as well as the argument and local variable number of the callee. Using the latter two operands, the LVP and CCP of the callee can be easily determined. Since the size of the caller context section is constant (currently equal to four), the SP can be calculated by simply adding the updated CCP to the size of the caller context section.

The caller context section contains the caller's SP, CCP, LVP as well as its AMTI and PC. The last two 16-bit values occupy a single entry. Note that the SP saved in the caller context section is equal to the LVP of the callee, rather than its original value. This is because the parameters of the callee need to be removed from the operand stack of the caller on return from the callee.

Once the invocation of the callee is complete, the three pointer registers are restored using the values held in the caller context section at first. After that, the result of the callee is pushed onto the operand stack of the caller. In the meantime, the AMTI and PC of the caller are sent to the token machine so that the caller can be executed further.

The bootloader method starts executing automatically after a system reset and does not have a calling method. Therefore, its stack frame needs to be additionally created on the Java stack of the main thread upon a system reset. This can be achieved by assigning predefined values to the three pointer registers. The LVP must be zero, because it points to the first entry of the entire Java stack. The CCP is currently set to 32 by default, allowing the stack frame to have up to 32 local variables. Accordingly, the default value of the SP is 36. The caller context section just holds four random values.

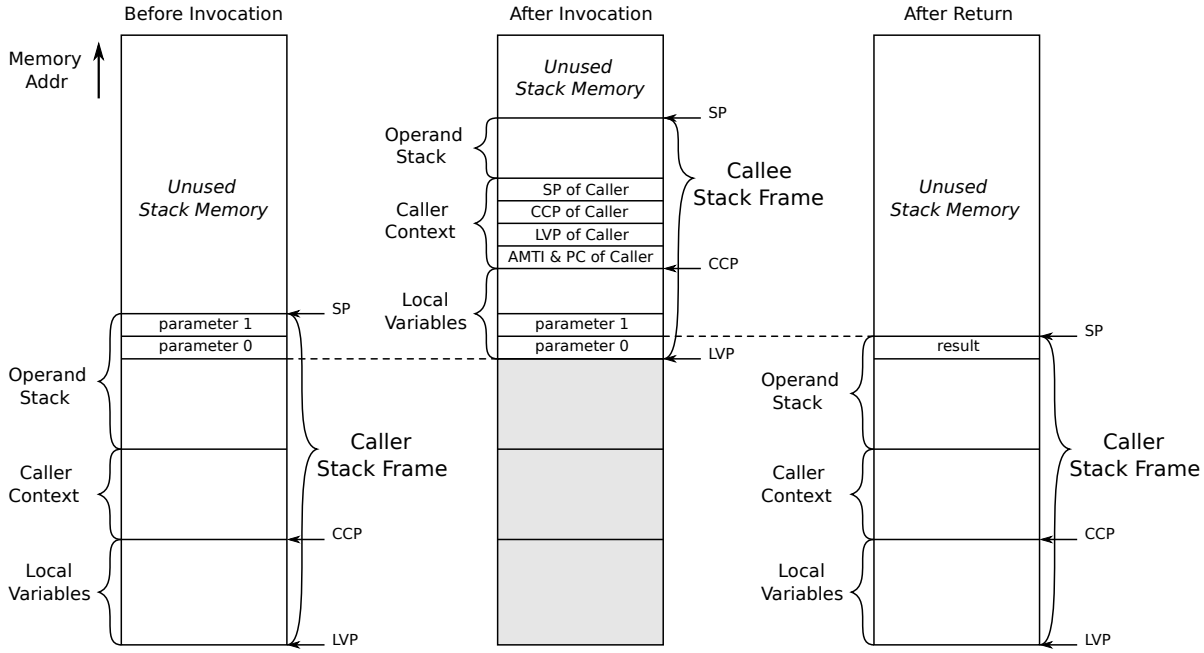


Figure 4.7: Stack frame creation and elimination

Thread Management

There is only one operation implemented for the purpose of thread management, namely `THREADSWITCH`. Upon a thread context switch, the frame stack receives the ID of the next thread from the token machine, as described in Section 4.3.5. On the one hand, it copies the current values of the SP, CCP, LVP, MP and overflow flag registers into the thread table, using the current thread's ID. On the other hand, it updates these registers with the corresponding values of the next thread. Exploiting the dual access ports of the thread table, both processes can be performed simultaneously. After all five registers have been updated, the ID of the next thread is written into the current TID register. Since the Java stacks of both current and next threads are resident in the stack memory, no other operation is needed. As a result, a context switch inside the frame stack takes 6 clock cycles totally.

4.4.3 Generation of Root Set

One of the key tasks of the frame stack is to assist with the garbage collection. It holds the major part of the root set, which consists of all object handles in the local variable array and operand stack of any stack frame. These object handles are required by the garbage collector at the beginning of the mark phase of a garbage collection cycle.

To allow an efficient communication, the frame stack provides a dedicated interface to the garbage collector. Once it receives a request from the garbage collector, the frame stack first completes its current operation. Then, it traverses the stack frames of all threads alive to find object handles, according to the type information associated with every stack entry. Each time an object handle is found, the frame stack notifies the garbage collector, using a $valid$-signal. This causes that the object handle is pushed onto an internal stack of the garbage collector, which is described in Section 4.5.5. After all object handles have been transfered, the frame stack sets a dedicated flag signal for a single clock cycle to inform the garbage

collector. In the current implementation, the frame stack cannot execute tokens as long as it generates the root set for the garbage collector.

4.4.4 Overflow Handling

As described in Section 4.3.4, a hardware exception handler is invoked by the token machine directly, if an exception is thrown by an FU at runtime. A special case that needs to be considered is the exception caused by a stack overflow. In this case, the hardware exception handler must be able to run on the current Java stack further which is, however, already full. To solve this issue, a Java stack may be assigned a set of additional entries that can only be used for executing the hardware exception handler. The number of these extra entries can be defined by using a configuration parameter.

4.5 Heap Manager

Memory for all class instances and arrays is allocated from the heap, as described in Section 2.3.1. A running Java application can create an enormous amount of objects of different size, without freeing them explicitly. Thus, there are two key aspects that need to be considered when designing the heap for a Java runtime system. First, the chosen size of the heap should be large enough to store all live objects. Second, a garbage collection mechanism is necessary to reclaim memory occupied by objects that are no longer referenced by the application code. Upon both aspects, several important design decisions were made for the heap of the AMIDAR processor as follows:

- Usage of off-chip DRAM to provide sufficient memory for holding all live objects.
- Implementation of a cache system to increase the heap access performance.
- Implementation of a hardware garbage collector that works autonomously without requiring the intervention of the processor.

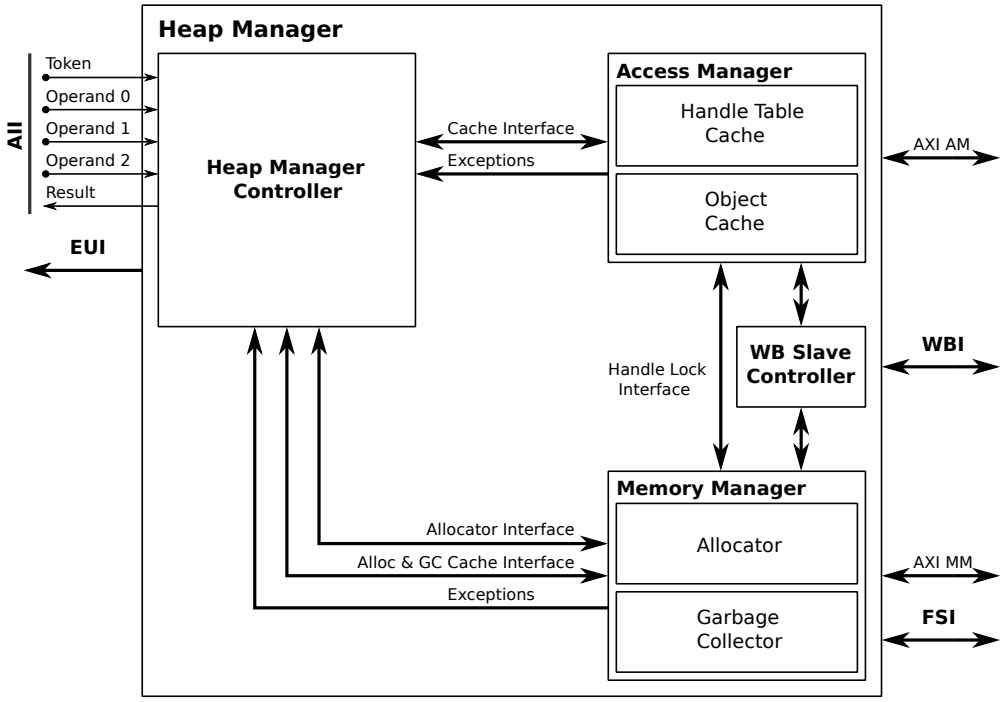
To meet the goals above, an FU referred to as *heap manager* below has been developed. It is constructed modularly and consists of an *controller*, an *access manager*, a *memory manager* as well as a *WB slave controller*, as shown in Figure 4.8.

In the following, Section 4.5.1 first introduces the layout of the off-chip memory holding the heap. After that, Section 4.5.2 provides a brief overview about each component of the heap manager. Section 4.5.3 describes the object cache in detail. In Section 4.5.4, the object allocation process is explained. Section 4.5.5 presents how the garbage collector works.

4.5.1 Memory Layout

The AMIDAR processor uses an external DRAM as its main memory that consists of a *boot data* area and a *runtime data* area, as represented in Figure 4.9. The former area contains data solely required during booting, while the latter one stores both static (e.g. the constant pool) and dynamic (e.g. objects) data used at runtime.

After a system reset, the AXT file of the Java application that needs to be executed on the AMIDAR processor is loaded at address 0 of the DRAM. Consequently, the whole boot data area is initialized, as



AII: *AMIDAR Infrastructure Interface* **EUI:** *Exception Unit Interface* **WBI:** *Wishbone Interface*
FSI: *Frame Stack Interface*

Figure 4.8: Heap manager overview

well as the first part of the runtime data area. Then, the bootloader method starts configuring the token machine, the heap manager as well as the frame stack by exploiting their WB interfaces.

To configure the heap manager, the bootloader method reads out the heap- and GC-specific values from the AXT header, like the offset of the info section or the offset of the heap, and writes them to the corresponding WB registers of the heap manager. After the heap manager has received all these values, an internal initialization process is triggered. During this initialization, the bootloader method stays in a busy-waiting loop and keeps polling a WB status register of the heap manager, which indicates if the initialization is complete.

The heap manager performs the initialization in two phases. It first loads the entire info section of the AXT file into a BRAM-based module of the garbage collector called *GCInfo*. During a garbage collection cycle, this module provides necessary information about a class according to the given CTI of the class. Then, the heap manager writes 0 across the rest of the main memory to ensure that any instance field will be assigned a correct default value. Finally, it asserts the status register to allow the bootloader method to proceed.

As Figure 4.9 illustrates, the runtime data area is made up of the method area and the heap, both of which belong to the standard Java memory model. The method area and its usage have been described in Section 4.2 and 4.3. Therefore, this section solely focuses on the heap. The heap can be further partitioned into five parts: an *immortal heap*, a *handle table*, a *WB handle table*, a *GC stack* and a *dynamic heap*. Since the immortal heap and the first part of the handle table are generated statically, they are marked in gray in the figure.

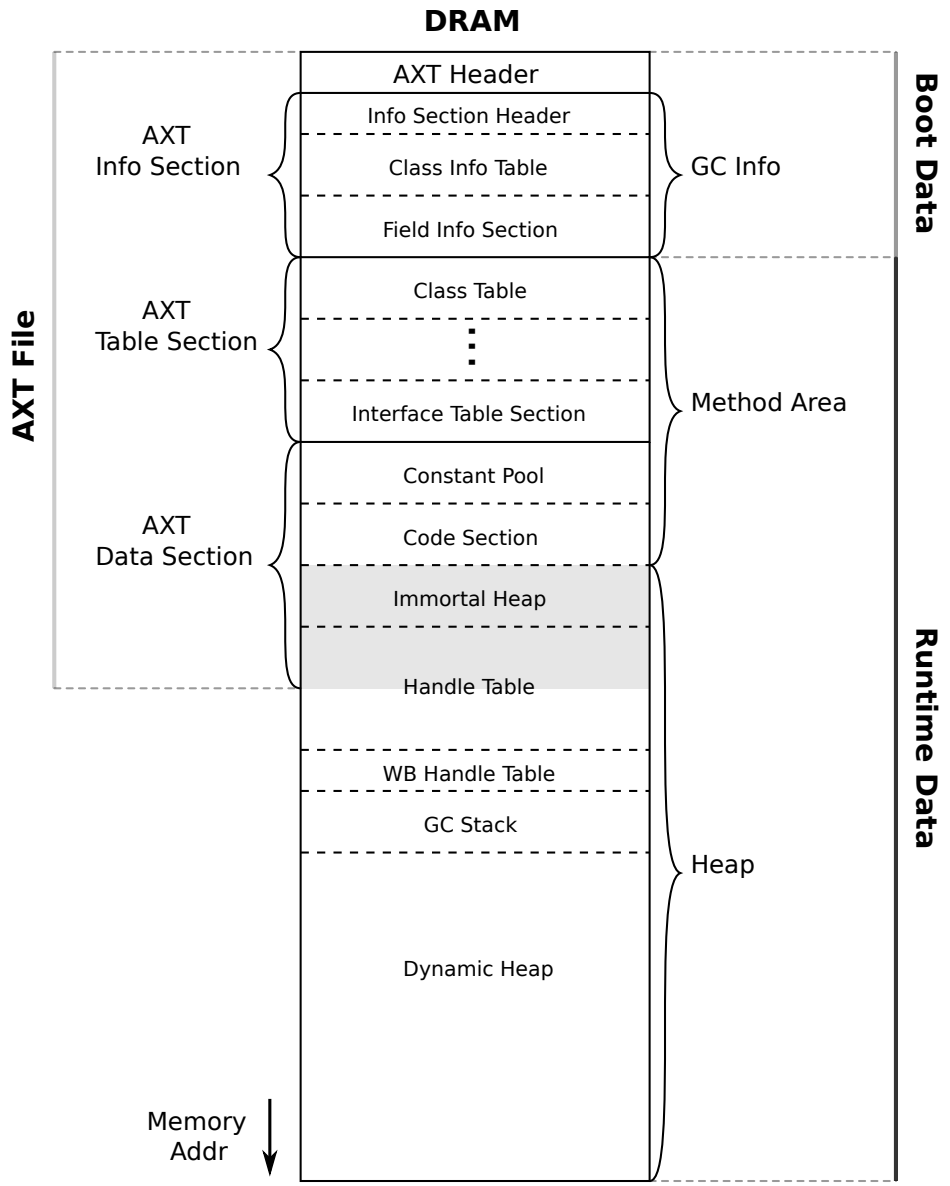


Figure 4.9: Memory layout

Immortal Heap

The immortal heap stores all pregenerated objects, including the static field object, a number of string objects and their char arrays, as well as the class objects of all classes contained in the AXT file. As its name suggests, the immortal heap is not cleaned up by the garbage collector at runtime.

Handle Table

The handle table holds the header of each object, whether immortal or dynamic. The header structure has been introduced in Section 4.2.5 above. The index of the header of an object is called the object's handle. The width of a handle is 32 bits. However, the maximum handle allowed is limited by the predefined size of the handle table. An object is addressed by its handle instead of its physical address inside the AMIDAR processor. Due to this, the reference to an object just means the handle of the object in the following description.

There are three handles that have special meanings for the memory manager. The first one is handle 0, which represents the null pointer. The second is handle 1, which is assigned to the static field object. The third is the last handle of the pregenerated handle table. Any handle that is equal to or less than it is associated with either an immortal object or the null pointer. Therefore, such a handle is ignored by the garbage collector in the mark phase, except handle 1. This is because the static field object contains an important part of the root set.

To speed up the access to the handle table, the access manager includes a handle table cache, which is described in Section 4.5.2 below. A cache miss causes the missing handle table entry to be loaded from the main memory into the cache. In this case, the cache controller needs to convert the given handle, H, to a byte-aligned physical address as follows:

$$Phy\_Addr_H = Base\_Addr_{HT} + H \ll 3 + H \ll 2 \quad (1)$$

where $Base\_Addr_{HT}$ represents the base address of the handle table in the main memory. This value is saved in a WB register of the heap manager.

WB Handle Table

As mentioned above, a hardware component, like an FU or a peripheral device, can be mapped to a WB object, which allows it to be accessed at the software level directly. A WB object is also assigned a unique handle so that it can be treated as a regular object. A WB handle differs from a normal handle in that its most significant bit (i.e. bit 31) is set to 1.

The WB handle table holds the headers of all WB objects. The header structure of a WB object is the same as that of a normal object. However, except the CTI, the other three header attributes are actually not used at runtime because a WB object is not allocated from the heap. The header of a WB object is accessed through the handle table cache included in the access manager as well. According to the handle of the WB object, the physical address of the header can be calculated upon a cache miss, using Equation 1 above. The only change is that the base address of the handle table needs to be replaced with that of the WB handle table. The WB flag bit of the handle is automatically removed via the left shift operation.

GC Stack

In the mark phase, the garbage collector requires a stack to assist with tracing the handles of live objects. Theoretically, this stack should be able to hold the handles of all live objects at the same time. To meet this requirement, it is placed into the DRAM and referred to as GC stack below. To reduce the access delay, the garbage collector contains a BRAM-based GC stack cache with support for the basic *spill-and-fill* protocol.

Dynamic Heap

The memory of all objects and arrays created dynamically is allocated from this area. Thus, it is the primary work space of the memory manager. The dynamic heap is divided into two equally sized semi-spaces. As the garbage collector compacts one semi-space, new objects can be allocated from the other

semi-space. This allows the garbage collector to run concurrently with the processor in the compact phase.

4.5.2 Components of the Heap Manager

Controller of the Heap Manager

The controller serves as a glue logic between the rest of the heap manager and the AMIDAR processor. Through the standard AMIDAR infrastructure interface, it receives tokens and their operands. However, the controller does not execute any operation by itself. It simply decodes an incoming token and then delegates the corresponding operation to either the access manager or the memory manager. After the operation has been completed, the controller redirects the result provided by either of both management units to the target FU directly.

The heap manager provides a small set of operations, which are described in Appendix B.3. These operations can be divided into four groups:

- Allocation (e.g. `ALLOC_OBJ`, `ALLOC_ARRAY`).
- Access to object field or array element (e.g. `HO_READ_OBJ`, `HO_WRITE_ARRAY`).
- Access to object header attribute (e.g. `GET_SIZE`, `GET_CTI`).
- Operations required by native functions (e.g. `PHY_READ`, `FLUSH`)

All operations above are executed by the access manager, except the allocation ones. During the execution of an access operation, two exceptions can be thrown by the access manager, namely **`NullPointerException`** and **`ArrayIndexOutOfBoundsException`**. At the end of a garbage collection cycle, the memory manager can throw either of the following errors: **`OutOfMemoryError`** or **`OutOfHandleError`**. On the occurrence of any of these exceptions or errors, the controller suspends the whole heap manager immediately and then signals the exception unit, using a dedicated interface (i.e. the EUI shown in Figure 4.8).

As Figure 4.8 illustrates, the access manager contains a handle table cache and an object cache. Both caches are shared by the controller and the memory manager. The controller uses them to carry out access operations delivered to it by means of tokens. The memory manager requires them in several different situations, which are described later in the following sections below. A simple example is that the garbage collector needs to update the physical address of an object after this object has been relocated in the compact phase. Since the controller and the memory manager might require the same cache simultaneously, a synchronization mechanism is needed to avoid race conditions. For this purpose, the cache access signals of the memory manager are connected to the controller rather than the access manager. The controller grants access permissions for the caches and prefers the memory manager over itself. Due to this central arbitration, the access manager only needs to provide a single access interface, which greatly simplifies its control logic.

Access Manager

As Figure 4.10 illustrates, the key components of the access manager include a handle table cache and an object cache. They aim to allow efficient access to object headers and fields. The access manager

can only perform a single access to either of these caches at a time. Upon a cache hit, both caches yield a result in a single clock cycle. Upon a cache miss, the access manager blocks until the missing data is loaded from the main memory.

The handle table cache and the object cache follow quite different access patterns, as discussed below. One of the key research goals of the AMIDAR project is to develop a speedy object cache based on the indirect object addressing scheme. Therefore, the implementation of the object cache is discussed in detail in the next section. This section only provides a brief overview about the handle table cache and the main differences between both of the caches.

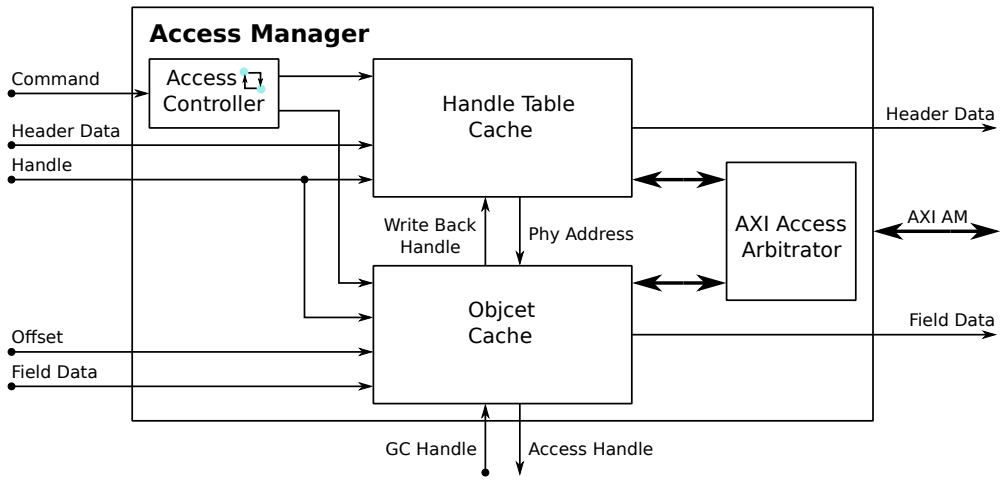


Figure 4.10: Access manager overview

All entries in the handle table are of the same size because of the fixed object header structure. Each of the four header attributes can be selected by using 2 bits integrated into the access command. Therefore, only the handle of an object is required to access the object's header. As a result, the handle table cache simply uses the least significant bits of a given handle as the cache index. In contrast, an object may contain an arbitrary number of fields. For this reason, to locate the position of a field inside the object, the offset of the field is also necessary in addition to the object's handle. This means that the index generation of the object cache must be performed based on both handle and offset. There are several different schemes proposed for this purpose, which have been discussed in Section 3.2. In Section 4.5.3, the cache index generation scheme used in the AMIDAR processor is described.

The handle of an object, i.e. the position of the object's header in the handle table, is completely independent of the physical position of the object in the main memory, especially when considering that the garbage collector reallocates objects in the compact phase. Consequently, the handle table cache can barely benefit from spatial locality. Thus, a single cache line of the handle table cache only contains one object header. Also, according to measurements of multiple benchmarks, the 2-way set-associative cache provides the best average hit rate under the condition that the cache size remains constant (currently 12 kilobytes). Unlike the handle table cache, the object cache can enjoy good spatial locality. To exploit the existing spatial locality and provide the best average hit rate, several important cache parameters need to be selected carefully, like the cache line size and the associativity. The next section presents a thorough discussion about the determination of these parameters.

Upon a cache miss, the controller of the handle table cache generates the physical address of the missing object header by using Equation 1 described in Section 4.5.1. From this address, a new cache line is loaded into the cache. In contrast, if some field of an object that needs to be accessed does not exist in the object cache, the cache controller first needs to read out the base address of the object from the handle table cache. Then, the physical address of the field is calculated by adding the field's offset to the object's base address. This implies that an access to the object cache may cause two cache misses in the worst case: one in the object cache itself, another in the handle table cache. However, both caches will never access the main memory simultaneously. Thus, both of them share a single AXI port.

The write policy used by the handle table cache is *write-through* combined with *non-write allocate*, which considerably simplifies the cache control logic. There are two major reasons for this design decision:

- After an object has been allocated, its header is read-accessed most of the time. Only the garbage collector may change the header attributes of an object, e.g. to update the object's physical address after it has been reallocated in the compact phase. Therefore, the latency incurred by the write-access to the main memory is negligible. Additionally, a FIFO-based buffer is employed to support *posted-write* so that a write access takes only one clock cycle as long as the buffer is not full.
- The garbage collector needs to read the reachability level of every object from that object's header in the compact phase. As mentioned above, in this phase, the processor runs in parallel with the garbage collector. If the garbage collector used the cache interface provided by the controller of the heap manager, it would delay access operations requested by the processor. To avoid this, the garbage collector accesses the handle table in the main memory directly. Due to this, the write-through policy is necessary to guarantee the data consistency between the cache and the main memory.

Unlike object headers, the vast majority of object fields must be updated continuously as a Java application is running. Therefore, the object cache uses a classical approach to speed up write accesses, namely *write-back* combined with *write allocate*.

The object cache provides an additional interface to the garbage collector, which is used to avoid object access collisions between them in the compact phase. An object access collision can happen if the object cache writes the data of some object back to the main memory due to a cache eviction, while the garbage collector is reallocating the object. To avoid this collision, either of these components must first check if the object is currently in use through sending the object's handle to the other component over the interface between them. If it receives an acknowledgement from the other component, it may perform the corresponding operation on the object. Otherwise, it has to wait until the operation that is being performed on the object is complete. This synchronization mechanism is quite similar to that used by Java, where the handle of every object can be considered as the object's monitor. During the compact phase, either of the object cache and the garbage collector must first obtain the monitor of an object, before it may perform a critical operation on the object.

Memory Manager

The structure of the memory manager is shown in Figure 4.11. This subsection briefly introduces the key components of the memory manager at the functional level and their important interfaces. On

this basis, Section 4.5.4 explains the allocation process in detail, while Section 4.5.5 describes how the garbage collector traces live objects and reclaims memory of dead objects.

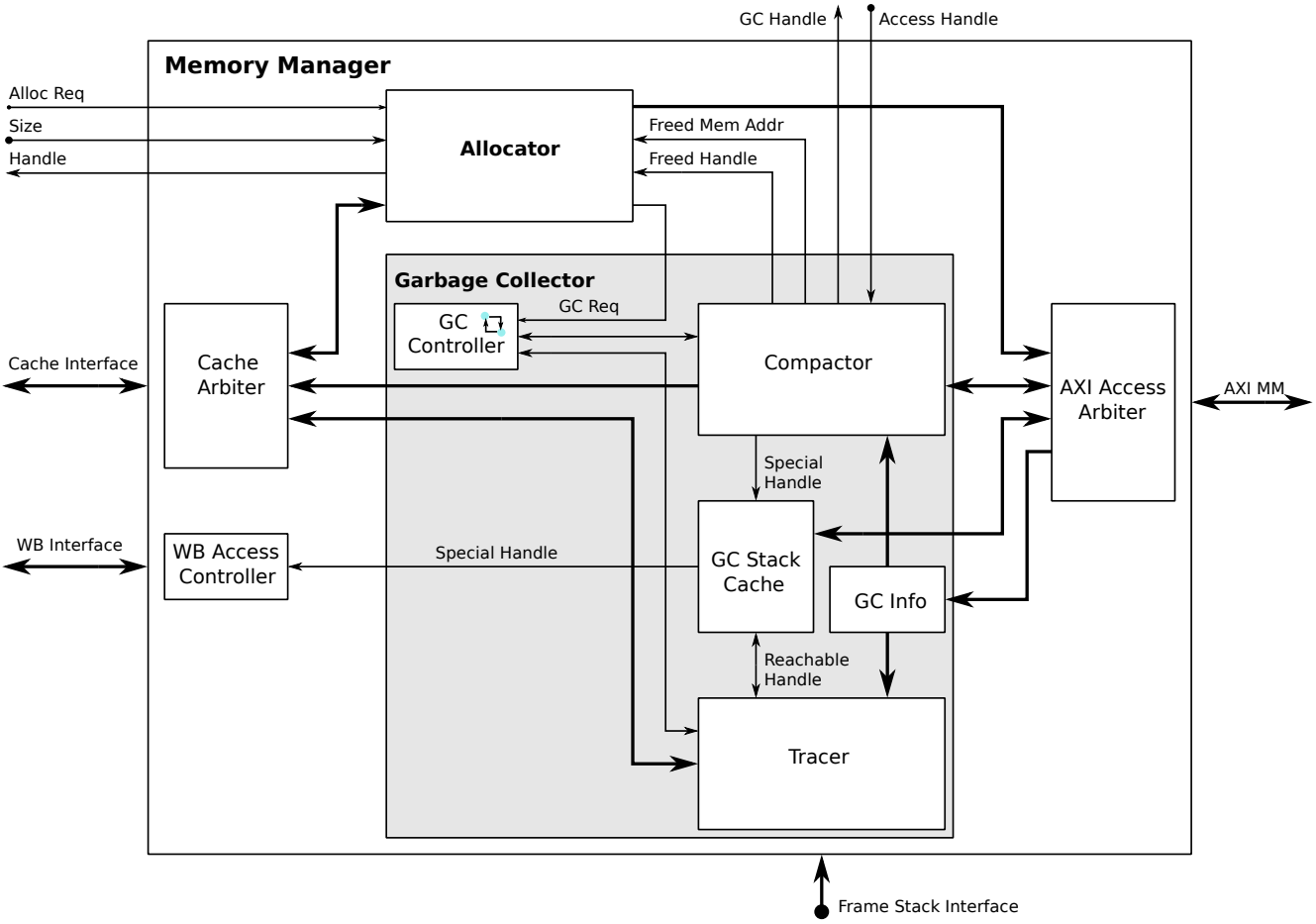


Figure 4.11: Memory manager overview

Allocator: Its main function is to allocate objects and arrays from the dynamic heap. Due to the indirect object addressing scheme used by the AMIDAR processor, two resources are required for this purpose: a free handle and sufficient memory on the dynamic heap. If both resources are available, the allocator will return the handle of the newly allocated object. Otherwise, the garbage collection process will be triggered. During the allocation of an object, the allocator must access the handle table cache twice: firstly, for reading the next free handle and secondly, for writing the physical address of the object into the object's header (for more details, see Section 4.5.4 below). During the garbage collection process, the allocator blocks until the mark phase is complete.

Garbage Collector: It is the centerpiece of the memory manager and stays inactive as long as there are enough resources for allocating a new object. As shown in Figure 4.11, the garbage collector can be further divided into the following submodules:

- **GC controller** coordinates the execution of the mark phase and the compact phase, which are performed by the tracer module and the compactor module respectively. It deactivates the compactor until the end of the mark phase.

- **Tracer** marks every live object and assigns a reachability level to it, using the GC-specific flags stored in the object's header. The current version of the tracer works in a stop-the-world manner, i.e. the processor is halted during the whole mark phase. Through a dedicated interface to the frame stack, the tracer loads the root set at the beginning of the mark phase, from which the graph of references is traversed. Each time after an object, *O*, has been marked as reachable, the tracer will read out the handle of each object referenced by *O* from the object cache and mark this object as reachable too.

- **Compactor** reclaims the handle table entries and the memory occupied by dead objects. As mentioned in Section 4.5.1, the dynamic heap is partitioned into two equally sized semi-spaces. While the compactor compacts one of them, the allocator can allocate objects from the other one. The freed handle table entries are linked together to form a list, where the physical address attribute of a free entry holds the index of the next free entry. As soon as the compactor is activated, it returns the handle table index of the first entry in the linked list and the basic address of the freed memory area to the allocator, which were reclaimed in the previous garbage collection cycle. This allows the allocator to start running already after the mark phase rather than the compact phase. After an object has been reallocated in the main memory, the compactor needs to access the handle table cache to update the object's physical address. If the object is of type **`SoftReference`** or **`WeakReference`**, the compactor also needs to access the object cache to clear the **`referent`** field of the object additionally, as described in Section 4.5.5 below.

- **GC info module** assists with executing both of the mark and compact phases. According to a given CTI, it can provide the following GC-specific information about the corresponding class:

  - If the class has at least a nonprimitive field.
  - The offset of every nonprimitive field of the class.
  - If the class has a nonempty **`finalize`**-method.
  - If the class is one of **`SoftReference`**, **`WeakReference`** or **`PhantomReference`**.

  The tracer requires the first two information so that it is able to exactly identify every reference. The compactor exploits the last two information to find out objects that need to be handled specially.

- **GC stack cache** is used by both of the tracer and the compactor. At the beginning of the mark phase, the tracer first pushes the handles included in the root set onto the GC stack through this cache. Then, these handles are popped from the GC stack one at a time. Each time the handle of an object *O* is popped, the handles of the objects referenced by *O* are pushed onto the GC stack in sequence. As the handle of an object is pushed onto the GC stack, the object is marked as reachable and is assigned a reachability level. Once the GC stack becomes empty, the mark phase is complete. In the compact phase, the compactor pushes the handles of special objects onto the GC stack. An object is considered to be special, if the object is unreachable and has a nonempty **`finalize`**-method, or the object is of type **`SoftReference`**, **`WeakReference`** or **`PhantomReference`** and has not been enqueued. The GC stack cache provides a WB interface through which the handles of special objects can be read out at the software level, allowing these objects to be handled in a dedicated thread.

4.5.3 Object Cache

The logically addressed object cache is one of the key components of the heap manager because all accesses to objects need to be performed through it. This indicates that its performance, i.e. its average hit rate, can affect the performance of the whole system significantly. This section presents a thorough discussion centered around the question of how to construct an object cache such that it can provide an optimal average hit rate.

Cache Basics and Terminology

A cache is usually a memory matrix that provides space for data. Each contiguous block of data is called a cache line or a cache block. The number of words contained in a cache block is called the block size, which is typically in the range of 4 to 16. Every word of a cache block can be addressed individually by its offset inside the cache block. A cache block requires additional administrative information including a tag, a valid flag and modified flags. The memory matrix is addressed by a set of bits that is called the index. The index is usually computed from the physical or virtual address.

As in normal RISC processors, data access does not exhibit as much spatial locality as instruction access. Thus, it is common to build the data cache as a set-associative cache providing multiple locations in cache that can be used to store data belonging to the same index. Each location in one set is called a way. The number of ways is also referred to as the associativity. The number of bits used to index the cache can be computed from the cache size, the block size, and the associativity as follows:

$$N_{index} = ld\left(\frac{cache\ size}{block\ size\ *\ associativity}\right) \tag{2}$$

Since one position in a cache can store many different physical memory locations, it is necessary to store which physical location is currently stored in the cache. This is done with the tag information. In terms of hardware effort, a larger block size requires less resources and larger associativity requires more resources to build a cache of the same size.

Logically Addressed Object Cache

In contrast to a physically or virtually addressed cache, an object cache is addressed by using an object handle and a field offset. Thus, the major problem in this case is the creation of an index for the cache from these two values. For the sake of simplicity, we only discuss a read access to a field of an object here. First, the object's handle and the field's offset are passed into the cache. Then, an index, a tag and a block offset are generated from them. The index is used to access one set in the cache. The tag information of the selected set is then compared with the generated tag information and if one tag matches, the corresponding cache block is forwarded to the word selection logic which evaluates the block offset.

The easiest way to create index, tag and block offset is a fixed bit selection. Figure 4.12 shows such an approach based on the third index generation scheme discussed in Section 3.2, namely concatenation of the LSBs of the handle with some bits in the middle of the offset. In this example, the block offset

is only composed of the 3 LSBs of the offset. The 9-bit index is composed of 8 handle bits and a single offset bit. All bits not used in the block offset or in the index have to be treated as tag information.

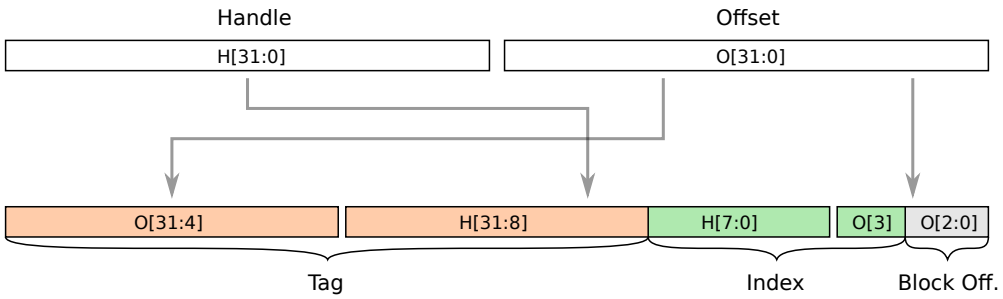


Figure 4.12: Set index generation using fixed scheme

As mentioned in Section 3.2, known object caches only use several LSBs of the offset (up to 12 bits) for the index generation and simply discard the remaining bits to reduce the memory overhead for maintaining the tag bits. This design decision is based on the observation that the majority of objects are small. However, such an object cache requires an additional mechanism for handling larger objects. This section aims to propose a general index generation scheme that is feasible in any situation. Therefore, the entire offset is employed to generate the index and tag in the example above.

Typically, one would use the LSBs of the handle for the index generation to ensure enough entropy for the index. In this way, objects are always aligned to cache blocks. Also, this means that only one object can reside in one cache block. If an object is much smaller than the cache block, a considerable part of this block is unusable.

Side Effects of Logical Addressing

To minimize the number of physical memory accesses, the cache is usually operated in the write-back mode. Together with the fact that only part of the cache block holds relevant object information, this can lead to additional problems. Once a cache block is written back to main memory, care must be taken to write only as many words of the cache block back to physical memory as the object actually occupies. Thus, the highest written offset within each cache block must be tracked by the cache, since it is not possible for the cache to evaluate the actual object size. Failure to do so would cause an overwriting of the following object in physical memory.

Since only a part of the index is determined by the field offset, this limits the amount of cache space that can be devoted to one particular object at a time. In a physically addressed cache, such a limitation does not exist. In the extreme case, the cache can be filled completely by one object. This is impossible in a logically addressed cache. The amount of cache blocks that can be used by one object is determined by the number of offset bits in the index. If no offset bits are part of the index, an object can occupy all ways of only one particular set. With every offset bit that is added to the index, this number doubles. Thus, a fixed selection is always a compromise and the characteristic of the running application may not suite the selection.

State of the Art Approach

As discussed in Section 3.2, previous research has already tackled the problem of index generation in object-based memory systems. The approach presented in [123] tries to overcome the problems of a fixed bit selection by mixing bits of the handle and bits of the offset by an XOR combination. The paper proposes to use 6 bits from each of the handle and offset.

The reasoning for this approach follows this line: small objects will have zero bits in the upper part of the 6 bits. Thus, for those objects the index is determined mainly by the handle. Large objects can still use a considerable part of the cache. In this case, a constant handle is combined with the varying upper bits of the offset. While this sounds reasonable at a first glance, we believe that it still does not support all possible situations.

Dynamic Mask Selection

The basic idea of our index generation scheme is to give up fixed bit selection. Instead, we provide different bit selections called masks. Small objects should use only a small part of the offset for the index generation, while large objects should use more bits of the offset. The number of different masks is an optimization parameter. Figure 4.13 shows the index composition for an index of 9 bits.

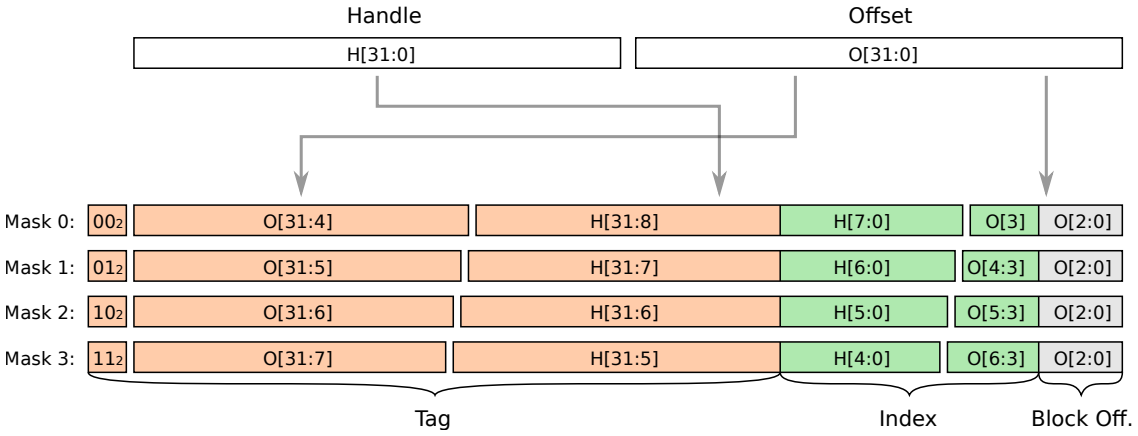


Figure 4.13: Set index generation using dynamic scheme

The remaining problem is the determination of the mask that shall be used for an actual access. As mentioned previously, the cache cannot determine the actual object size for a given access. Thus, in a first try we did not actually use the object size, but rather the offset used for the current access. We simply have to find the highest bit which is set to one in the offset. This information is used to select the mask. Higher bit values lead to masks that devote more index bits to the offset. Smaller bit values lead to masks that devote more index bits to the handle. In the example above, mask 3 will be selected if $O[6]$ is equal to 1_2, mask 2 if $O[6:5]$ is equal to 01_2 and so on.

Note that this scheme makes it impossible to reverse the operation and calculate the original handle and offset bits from a given index, which is, however, necessary for a write-back operation. Thus, each mask is assigned an identifier (e.g. the ID of mask 3 is 11_2) which needs to be stored in the tag information to check the exact match for a cache block and to provide the full address for a write-back operation. The number of bits required for keeping the mask identifier is determined as such:

$N_{mask\_id} = ld(\#masks)$. In the example above, up to 8 different masks can be used for the index generation, from $\{H[0], O[10:3]\}$ to $\{H[7:0], O[3]\}$.

Further Performance Analysis

In order to improve the index generation scheme, we carried out an analysis of the statistical properties of the individual index bits. Since all cache evaluations were done in a cache simulator, it was quite easy to evaluate the distribution of the index bits for a particular application. Table 4.17 shows exemplary statistics of jack from the SPEC JVM98 benchmark suite run by using the following cache configuration: cache size 64 kilobytes, block size 8 words, associativity 2.

Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0.54	0.48	0.48	0.43	0.55	0.67	0.38	0.43	0.65	0.64

Table 4.17: Bit probabilities of the individual index bits

It can be seen that particularly for LSBs of the index, the probabilities of 1 and 0 are not equally distributed. Naturally, this leads to imbalanced usage of the cache sets. Thus, it would be desirable to improve the index generation such that all index bits are equally distributed.

A more thorough analysis reveals that this imbalance is dependent on the application and the cache configuration. Running the same application with a larger block size, yields much better distribution of the index bits. Also, different applications might work well with the original configuration. Therefore, we looked into the statistics of object usage in this application. It turns out that a large number of objects has a size of 10 words. Let us assume that all fields of the object are used in equal manner. For a block size of 8 words, this means that offset bit 3 determines which set in the cache is used. If this bit is 0, all 8 words of the cache block can (and will) be accessed. If this bit is 1, only the first two words of this cache block will be used. This behavior explains the imbalance in the index bit distribution.

Further Optimization Attempts

We tried to improve the index generation in two ways. First, we tried to use the real object size instead of the current offset for the mask selection. Then, we tried to rectify the imbalance of the index bit distribution. For these purposes, several handle bits were used to hold additional object information. In a regular JVM, the handle includes at least 32 bits. In practice, it is highly unlikely that an application will use 2^{32} objects at the same time. Thus, we can use some of the handle bits to save additional information about the object being referenced.

To improve the mask selection, the three highest handle bits were employed to hold the mask identifiers preselected by the object allocator. This implies that up to 8 different masks can be adopted. The object allocator, which is in charge for the management of the handles knows the exact size of the object and thus, it can encode this information in the handle. Obviously, these bits have to be discarded before looking up the handle in the handle table.

To improve the imbalance, we added another two bits of information to the handle. One bit specifies whether the referenced object is unsuitable for the cache block size, i.e. whether the following relation holds for the size of the object: $block\ size < object\ size <= 1.5 * block\ size$. If this is the case, the

second bit is used to replace index bit 0. This gives the object allocator the freedom to put such objects in alternating sets of the cache.

Performance Evaluation

For our performance measurements, we used the SPEC JVM98 benchmarks and two real world programs from the field of embedded systems: an Ogg Vorbis decoder [54] and a VP8 decoder [51]. Note that SPEC JVM98 has been retired with the release of SPEC JVM2008. However, we wanted to keep the comparison to the XOR-based approach [123] that was evaluated by using SPEC JVM98 as fair as possible. As such, we decided to choose the old benchmark suite rather than the new one.

Since implementing the different cache variants in hardware would have taken a considerable tool runtime, we simulated the different cache variants with our own cache simulator. This cache simulator takes memory access traces generated by the AMIDAR simulator [6] as input. Eventually, after identifying the best cache configuration for all benchmarks, we implemented this version in hardware, where we use performance counters to record the cache access statistics. The miss rate measured on the hardware slightly differs from the simulated value. This can be attributed to the startup and garbage collection behaviors of the AMIDAR processor, which do not occur in the simulation environment.

We simulated the XOR-based approach together with the four proposed alternatives listed below:

- *DMS1*: dynamic mask selection based on field offset.
- *DMS2*: dynamic mask selection based on object size (encoded in handle).
- *DMS3*: dynamic mask selection with correction for unfavorable objects (encoded in handle).
- *DMS4*: combination of *DMS2* and *DMS3*.

We chose three different cache sizes (only considering the data plane, the tag information consumes additional storage): 16 kilobytes, 32 kilobytes, 64 kilobytes. Although we could easily simulate and generate cache implementations with larger data planes, we did not consider such huge L1-caches reasonable. For each cache size, we varied the associativity between 2 and 4. Also, we varied the block size between 4, 8 and 16 words. The cache replacement policy used was always pseudo-LRU. This gave us a total of 18 configurations for each approach.

One often discussed aspect of evaluating a cache is the *average memory access time* (AMAT) [45, 120] that can be calculated using the equation:

$$AMAT = 1 + mr \cdot t_{mem} \tag{3}$$

where mr represents the miss rate and t_{mem} the number of clock cycles needed to access an external memory like DRAM. Since t_{mem} heavily depends on the technology used, we just focus our discussion below on the miss rate.

In order to gain an overview on the quality of the DMS-based and XOR-based approaches, Table 4.18 provides average miss rates of the five approaches, calculated from results of running all benchmarks. The marked cell represents the best result of a cache configuration. The row of the best cache

configuration for each cache size is marked in gray. In these cases, DMS1 always performs best and is about 6 % better than the XOR-based approach.

Cache Configuration	DMS1	DMS2	DMS3	DMS4	XOR6	Δ% (XOR6 ⟷ DMS1)
16 KB, 4 words, 2 ways	50.91	51.95	51.80	50.81	53.80	5.37
16 KB, 4 words, 4 ways	37.33	43.93	43.79	37.33	43.09	13.35
16 KB, 8 words, 2 ways	35.78	43.45	43.10	36.55	39.08	8.43
16 KB, 8 words, 4 ways	31.42	33.64	33.29	31.87	33.40	5.95
16 KB, 16 words, 2 ways	36.14	42.78	41.13	37.66	37.86	4.54
16 KB, 16 words, 4 ways	33.30	34.70	34.53	34.57	33.08	-0.67
32 KB, 4 words, 2 ways	39.71	45.51	44.55	39.98	47.56	16.51
32 KB, 4 words, 4 ways	29.71	36.44	36.41	29.70	37.64	21.05
32 KB, 8 words, 2 ways	27.59	34.24	34.29	27.35	32.56	15.26
32 KB, 8 words, 4 ways	23.49	25.84	25.72	23.64	24.99	6.03
32 KB, 16 words, 2 ways	27.61	32.67	32.34	27.85	28.41	2.83
32 KB, 16 words, 4 ways	24.44	24.84	24.40	24.86	25.27	3.27
64 KB, 4 words, 2 ways	36.20	41.87	41.44	36.40	44.33	18.32
64 KB, 4 words, 4 ways	26.69	33.27	33.18	26.62	35.38	24.55
64 KB, 8 words, 2 ways	22.21	29.74	29.55	22.34	28.15	21.11
64 KB, 8 words, 4 ways	18.12	20.23	20.12	18.23	21.21	14.59
64 KB, 16 words, 2 ways	20.01	25.05	24.91	20.13	22.66	11.70
64 KB, 16 words, 4 ways	17.39	17.66	17.49	17.60	18.42	5.58

Table 4.18: Average miss rates for all cache configurations

Figure 4.14 shows the relative improvement of the miss rate for all benchmarks and all cache configurations for the best DMS-based approach (i.e. DMS1) in each case. The improvement is calculated as follows: $\left(1 - \frac{mr_{DMS}}{mr_{XOR}}\right) \cdot 100\ \%$. Positive bars mark a decreased miss rate compared to the XOR-based approach while negative bars stand for an increased miss rate. The number on the bottom of each box denotes the number of cache misses per 1000 read accesses for the DMS-based approach. As we can see, the DMS-based approach provides significantly better results for JOrbis and mpegaudio, and just slightly better results for compress and VP8Dec. For all other benchmarks the performance is the same or slightly worse.

A hint why our approach performs like this can be found in Table 4.19. For all cases that provide a good performance (marked cells), less than 50 % of all heap accesses are accesses to objects or arrays with a size smaller than 8 words. In all other cases, the percentage is more than 50 %. Thus, DMS can not unfold it's potential because most objects easily fit into one cache block. Note that not only the object size influences the miss rates alone but also the access patterns.

We conclude that our approaches have no considerable disadvantage compared to the XOR-based approach while in some cases the performance is significantly better.

Hardware Implementation

Based on the performance analysis above, we have implemented a 4-way set-associated object cache of 64 kilobytes in size, with 8-word cache blocks. According to Equation 2, this cache has 9 index bits,

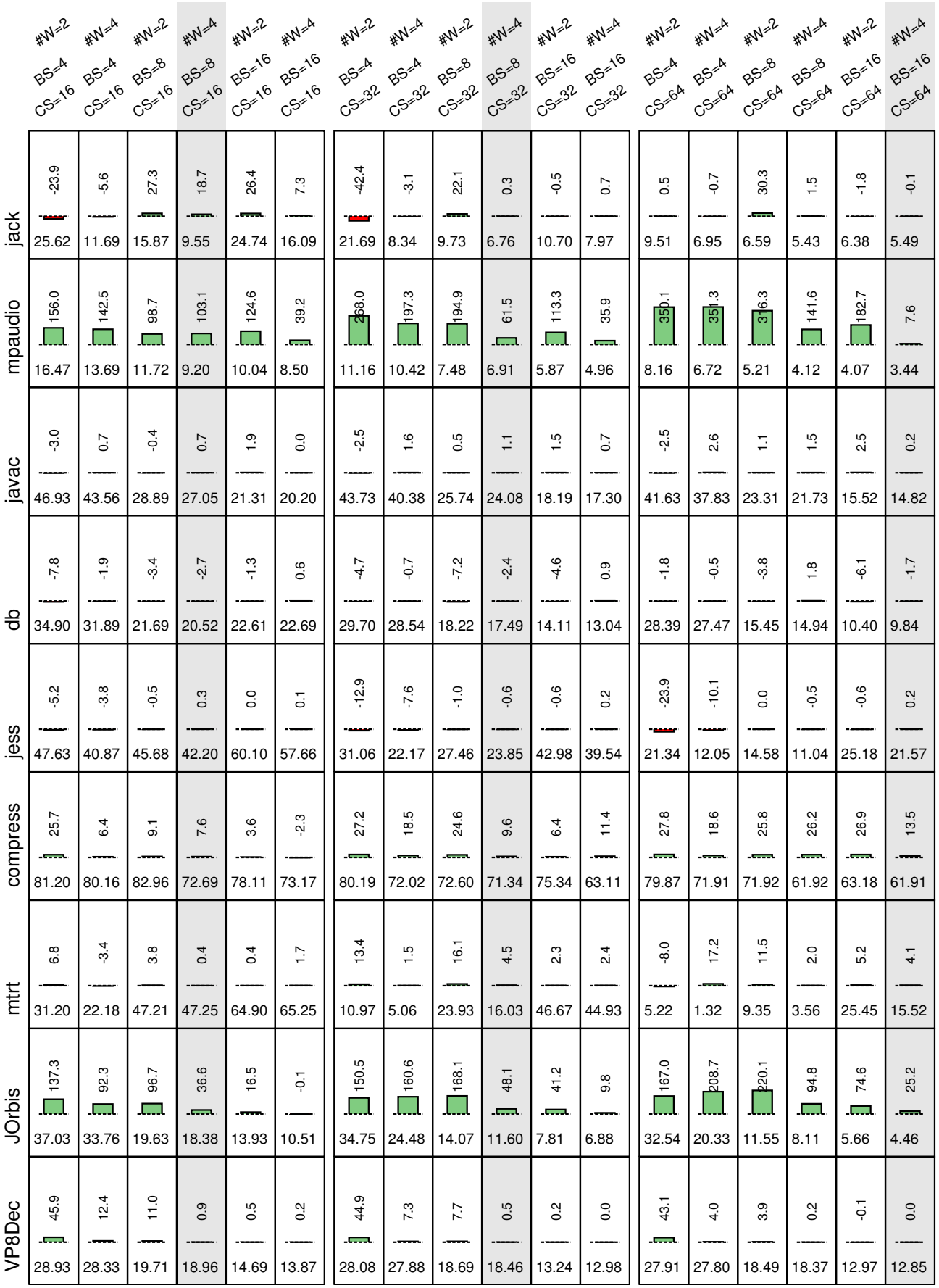


Figure 4.14: Miss rates for all applications with all cache configurations

jack	mpegaudio	javac	db	jess	compress	mtrt	JOrbis	VP8Dec
56.2 %	32.2 %	74.6 %	52.2 %	68.0 %	39.6 %	86.8 %	26.0 %	6.0 %

Table 4.19: Percentages of heap accesses to objects/arrays with size less than eight words

i.e. a total of 512 sets. Although using a block size of 16 words would provide a better miss rate for this cache (see Table 4.18), the AMAT would become longer regarding Equation 3, reducing the performance of the entire system. Figure 4.15 shows the structure of the cache, which includes an *index generator*, a *handle generator* and a general-purpose data cache module.

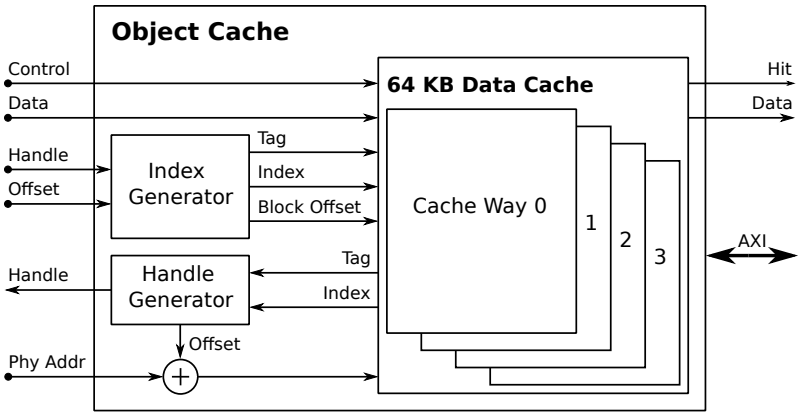


Figure 4.15: Object cache

The major task of the index generator is to create the cache index from the incoming handle-offset pair. To avoid an additional clock cycle delay, it solely includes a small combinational circuit for this purpose. The structure of this circuit is described at the end of this section below. The handle generator can reconstruct the handle and the offset of the first word contained in a cache block from the tag and the index of the cache block. Through the reconstructed handle, the base physical address of the corresponding object can be retrieved from the handle table. This is necessary for calculating the physical address of the cache block when it needs to be written back to the main memory. The data cache module provides the fundamental cache functions and can be customized to meet different requirements. Its replacement policy is pseudo-LRU, and its write policy is write-back combined with write allocate. Unlike a conventional data cache, it additionally saves for each cache block the offset of the last valid word into the tag information. This ensures that the data of the following object will not be overwritten by a write-back operation. To reduce the memory overhead for maintaining the tag information, the three highest bits from each of the handle and the offset are used to hold the mask identifier and the last valid offset of a cache block. These bits will be removed by the handle generator upon a cache eviction.

To simplify cache index generation in hardware, handle bits are arranged in reverse order inside an index, as shown in Figure 4.16. Through this arrangement, index bit 0 and 8 can be connected to offset bit 3 and handle bit 0 directly, without requiring any hardware logic. Each of the remaining index bits only needs to be selected between an offset bit and a handle bit, where the selection signal can be generated using a small chain of OR gates. Figure 4.17 illustrates the combinational circuit designed for this purpose.

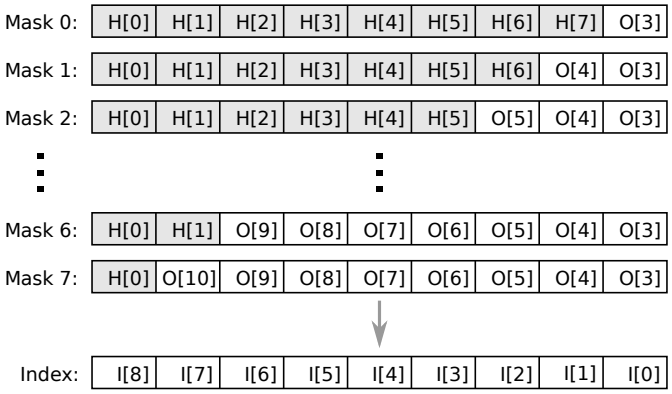


Figure 4.16: Index bit arrangement

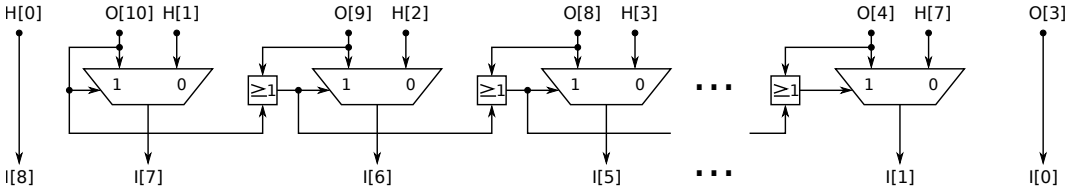


Figure 4.17: Index bit selection circuit

4.5.4 Object Allocation

There are two necessary resources for allocating a new object: a free handle and sufficient memory on the dynamic heap. The garbage collector establishes and maintains two linked lists to manage both resources, as shown in Figure 4.18.

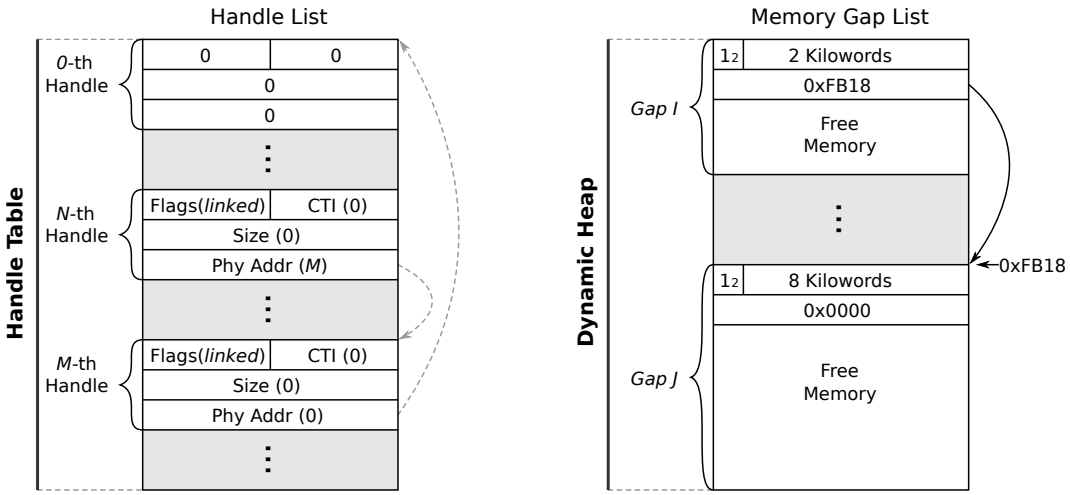


Figure 4.18: Handle and memory gap lists

Handle List

This list is included inside the handle table and consists of unused handle table entries. The physical address attribute of an entry contained in the list holds the index of the next unused entry, i.e. the next free handle. The tail of the list is linked to handle 0, namely the null pointer. Since the entries in the list are linked together logically via their indexes rather than their physical addresses, the links between

them are represented by gray, dashed lines in Figure 4.18. The logical linking simplifies traversing the handle list through the handle table cache that is logically addressed as well.

Once an object is no longer reachable, its handle table entry (i.e. the object's header) is reclaimed and attached to the end of the handle list by the garbage collector. If the list has not been established yet, this entry becomes the head of the list. Using a single-bit flag, this entry is marked as *linked*, indicating that its index is an available handle and may be assigned to another object.

As explained in Section 4.5.1, the heap manager writes 0 across the entire heap except the pre-generated part loaded from the AXT file during booting. This implies that the handle list actually does not exist until the first garbage collection cycle is complete. Thus, the allocator supports two different allocation modes: one uses the handle list, while the other not.

Memory Gap List

The dynamic heap is partitioned into two equally sized semi-spaces for the purpose of concurrent heap compaction. While one semi-space is being compacted, objects can be allocated from the other one in parallel. The garbage collector compacts a semi-space in such a way that it moves live objects over free memory space towards the beginning of the semi-space. This results in a large contiguous free memory area at the end of the semi-space. However, the AMIDAR processor allows programmers to lock objects on the heap to enable safe and speedy DMA transfers. A locked object (typically, an I/O buffer) may not be moved and therefore splits the free memory area into two parts, if it happened to be allocated in the middle of the area. To manage such discrete free memory areas efficiently, they are linked together by the garbage collector to form a list.

An entry contained in this list is a chunk of contiguous zeroed memory, which is referred to as *memory gap* below. A memory gap has a 2-word header. The first word denotes the beginning of the gap. The most significant bit of the word is set to 1 and the remaining 31 bits store the size of the gap in words. The second header word is a forwarding pointer holding the base address of the next gap, which means that the entries in this list are physically linked. For this reason, the link between the two gaps shown in Figure 4.18 is represented by a solid line. The forwarding point of the last gap is set to 0, indicating the end of the list.

Both of the semi-spaces of the dynamic heap have an individual gap list that may not cross the boundary between these semi-spaces. At the end of the heap initialization phase, i.e. after writing 0 across the heap, the heap manager creates in each of these semi-spaces a single memory gap consisting of the memory of the entire semi-space. This indicates that the initial memory gap list of a semi-space contains only one gap. If no object needs to be locked when executing an application, both of the memory gap lists will contain a single gap throughout the lifetime of the application.

Object Structure

An object is composed of a header and a contiguous memory block. The header is just a handle table entry. It holds the meta-information about the object, including several status flags, the CTI of the class from which the object has been created, the size and physical address of the memory block assigned to the object. If the object is created at runtime, its header is marked as *valid* by using a single-bit flag. In contrast, this flag of the header of an immortal object is not asserted, distinguishing the object from

those allocated dynamically. As the *valid* flag of a handle table entry is set, its *linked* flag is cleared at the same time, indicating that the corresponding handle is not free anymore.

The memory block of an object stores the object's fields. In the current implementation, a field of type **`long`** or **`double`** occupies 8 bytes, while a field of any other type occupies 4 bytes. This implies that all objects on the heap are word-aligned. To assist with the heap compaction, each dynamically allocated object is assigned an extra integer field by the allocator, which stores the object's handle. The offset of the field is -1 and therefore is not accessible at the software level. Figure 4.19 demonstrates the structure of a newly created object which has two 32-bit fields.

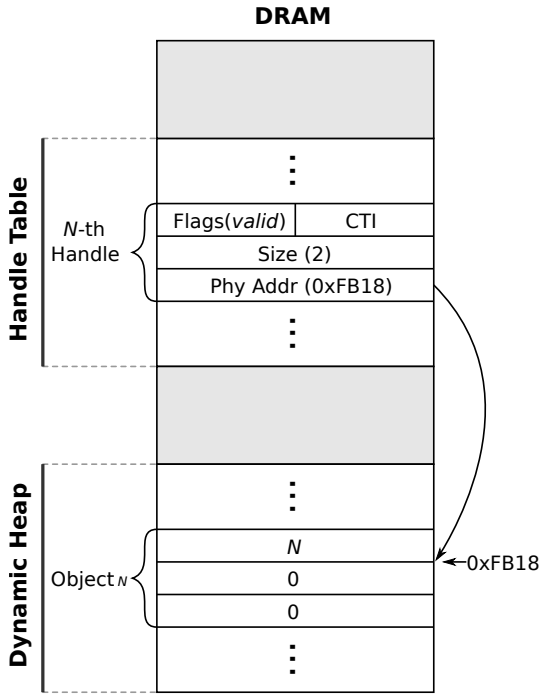


Figure 4.19: Internal object representation

An object needs to be allocated in two steps due to its structure. First, its header is allocated from the handle table, and then its memory block from the dynamic heap. In the following, the object allocation process is described in detail. To simplify the representation, the semi-space of the dynamic heap from which memory for objects is allocated is referred to as *allocation space*, while the other semi-space as *compaction space*. The memory gap being used in the allocation space is called the *current gap*.

Header Allocation

After a system reset, the handle list is still not established. In this phase, the allocator employs a counter to determine the next handle available. This counter is initialized with the size of the pregenerated handle table. Upon an allocation request, the allocator simply yields the current value of the counter as the handle of the new object and then increments the counter by 1. Once the counter exceeds a predefined threshold (by default, seven-eighths of the maximum handle number), the garbage collection process is triggered.

A new handle list is created in every garbage collection cycle. The head (i.e. the first handle) and size of the list are buffered inside the garbage collector and returned to the allocator as desired. If this list is the first one created after a system reset, the allocator will switch its allocation mode from the one

described above to the one using the handle list. If a handle list is already in use, the allocator simply attaches the new list to it.

To support the header allocation mode based on the handle list, the allocator requires two internal registers: one for tracking the current list head and the other for saving the initial size of the list. When allocating an object, the allocator assigns the list head to the object directly. Then, the next free handle is read out from the handle table cache and written into the list head register. Using a decremental counter, the allocator tracks the number of the remaining handles in the handle list. If the value of the counter falls below a given limit, a new garbage collection cycle is started. As a new handle list is attached to the existing one, both of the list size register and the decremental counter are updated with the sum of the current value of the decremental counter and the size of the new handle list.

Memory Allocation

Memory for objects is solely allocated from the current gap, until the gap becomes full or does not contain sufficient space for a new object. This causes that the size and base address of the current gap vary continuously. To avoid updating and relocating the gap's header in the main memory, which would be time-consuming, the allocator employs three registers to buffer the size, forwarding pointer and base address of the current gap. As a memory gap becomes the new current gap, the former two registers are initialized using the gap's header values, while the latter register is set to the forwarding pointer of the previous gap. Then, the header of the new current gap is zeroed in the main memory, allowing both of the header words to be assigned to a new object.

To allocate memory for an object, the allocator first checks if there is still sufficient space in the current gap. If this is the case, a memory block of the object's size is allocated at the beginning of the current gap. The physical address of the memory block (i.e. the base address of the current gap) is written into the object's header via the handle table cache. After that, the registers buffering the size and base address of the current gap are updated according to the object's size. If the current gap is not large enough for holding the object, the allocator creates a new header for the current gap in the main memory through writing the buffered size and forwarding pointer at the buffered base address. After that, the allocator skips to the next memory gap and attempts to allocate memory for the object in that gap. If the current gap is already the last gap of the allocation space, the garbage collection process is triggered.

A key thing to note about the memory allocation process is that the allocator traverses the memory gap list in a single direction. This means that the allocator will not return to the beginning of the list if the last gap cannot provide sufficient memory for a new object. This considerably simplifies the control logic of the allocator and therefore reduces the hardware usage.

4.5.5 Garbage Collection

This section first provides a quick overview of the garbage collection process, using a simple example. Then, key design decisions that were made for the garbage collector are discussed. After that, the implementation of the garbage collector is described in detail.

Sample Garbage Collection Cycle

The dynamic heap is partitioned into two independent semi-spaces, which allows the allocator to run in parallel as the garbage collector reclaims the memory occupied by dead objects. Figure 4.20 provides two sample snapshots of the dynamic heap that show the status of these semi-spaces before and after a garbage collection cycle.

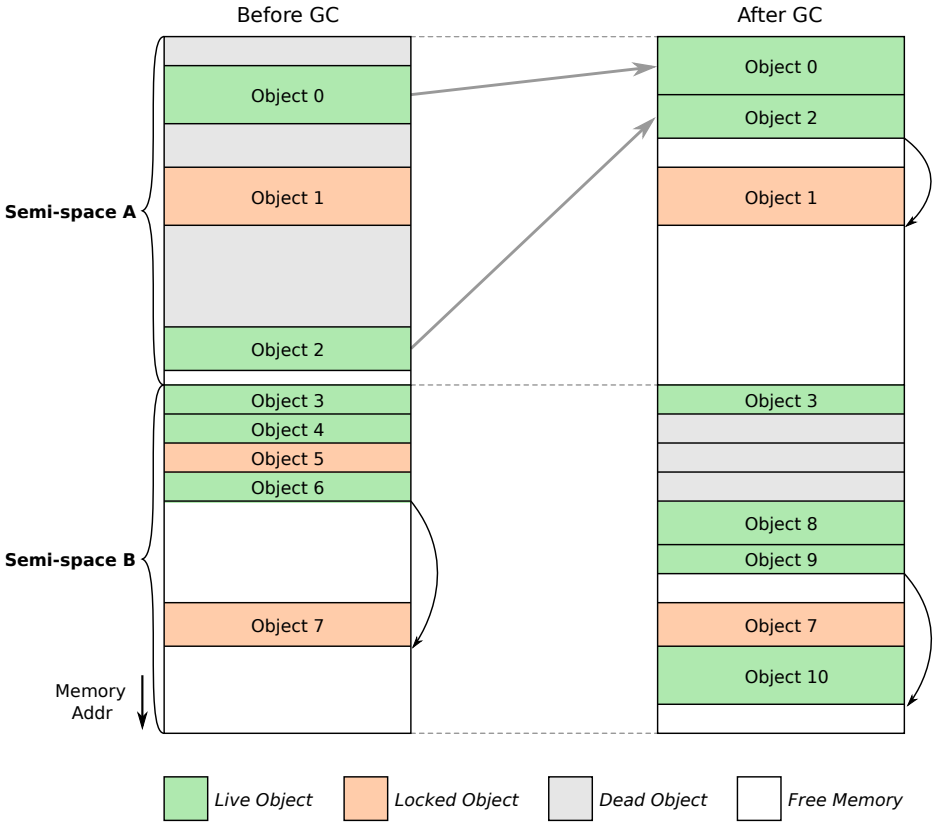


Figure 4.20: Snapshots of the dynamic heap

In this example, semi-space A serves as the allocation space at first. Accordingly, semi-space B is the compaction space and has already been cleaned up in the previous garbage collection cycle. The free memory contained in semi-space B is split into two gaps by object 7, which make up together a linked list. As the allocator attempts to allocate object 8 that cannot fit into the remaining free space of semi-space A, the garbage collection process is triggered. At this moment, semi-space B becomes the new allocation space and semi-space A turns into the compaction space.

During the mark phase, the garbage collector does a tree traversal of the entire root set and marks every object that it encounters as live, while the allocator simply blocks until the end of this phase. As soon as the garbage detection is complete, the allocator retries to allocate object 8 from semi-space B and succeeds. Then, it attempts to allocate object 9 and 10 in sequence. However, there is not enough free memory for object 10 in the current gap after the allocation of object 9. Thus, the allocator skips to the next free gap and accomplishes the requested allocation there. The free memory left in the former gap will not be used until the next garbage collection cycle.

In the meantime, the garbage collector compacts semi-space A by sliding live objects over free memory space towards the beginning of the semi-space. This compaction results in two free memory

gaps due to locked object 1. If a locked object becomes unreachable, like object 5, it will be removed automatically, without the need for explicitly unlocking it.

Design Decisions

According to the specific characteristics of the AMIDAR processor and the requirements of the Java specification, three design decisions were made, upon which the garbage collector has been developed.

1. Hardware-software collaboration for garbage collection.
2. Treating instances of **`SoftReference`** and **`WeakReference`** in the same way.
3. Combination of *stop-the-world* reference tracing and concurrent heap compaction.

Based on design decision 1, the whole garbage collector is made up of three components: the tracer, the compactor and a GC-specific thread which is simply called the GC thread below. As mentioned above, the former two components are implemented in hardware completely. They perform the two major tasks of the garbage collector, namely the garbage detection and the heap compaction. The GC thread is employed to execute two special operations: object finalization and enqueuing objects of type **`Reference`**. There are two major reasons for this design decision. First, execution of both operations in hardware would require multiple changes to the structure of the existing processor, for example, to enable invoking a method from the heap manager directly. This would considerably increase the hardware usage and unduly complicate the control logic of the heap manager and the token machine. Second, the Java specification does not explicitly define in which sequence and by which thread these operations should be executed, providing the maximum flexibility for the development of a Java runtime system.

According to the Java specification, if the garbage collector encounters a softly reachable object, it may choose to clear all soft references to that object at a time to make the object eligible for garbage collection. This means that the garbage collector is allowed to determine by itself when to perform the clear operation. It must only ensure that all soft references to softly reachable objects have been cleared before the runtime system runs out of memory. In contrast, all weak references to a weakly reachable object must be cleared atomically as soon as the object is encountered during the garbage collection process. The garbage collector of the AMIDAR processor treats a softly reachable object just like a weakly reachable one, i.e. it clears all soft references to the object when it encounters the object for the first time. The goal of this design decision is to avoid extra hardware logic that analyses the current memory usage, while still to obey the requirements defined in the Java specification.

The primary reason for the third design decision is that tracing references in a stop-the-world manner simplifies debugging tremendously, which is one important concern of hardware development. For our unit tests, the reference graphs at certain time points can be generated statically. This allows a straightforward determination of whether the tracer marks live objects correctly at these given time points. The secondary reason is that the vast majority of objects are short-lived and will not survive one garbage collection cycle [14, 127], which means that they will never be encountered in the trace phase. With regard to another fact that the heap compaction runs concurrently with the application, which is much more time-consuming than the reference tracing, the pause caused by the garbage collection process should be acceptable in most application scenarios.

In the following, the flags included in the object header are first presented, which are used by both of the allocator and the garbage collector. Then, the two working phases of the garbage collector are described respectively. Finalization of objects and handling objects of type **`Reference`** are introduced at the end of this section.

Header Flags

The header of an object includes 9 flags used for memory management, as shown in Table 4.20. Since most of these flags are self-explaining, only 3 of them are described below.

Flag	Width	Description
`Reachability_lvl`	3-bit	Reachability level of the object.
`GC_Cycle_ID`	2-bit	ID of the GC cycle in which the reachability level above is assigned.
`Enqueued`	1-bit	Asserted if the object is of type **`Reference`** and has been enqueued.
`Finalized`	1-bit	Asserted if the object has been finalized.
`Locked`	1-bit	Asserted if the object is locked.
`Valid`	1-bit	Asserted if the object is created at runtime.
`One_dim_prim_array`	1-bit	Asserted if the object is an one-dimensional primitive array.
`Array`	1-bit	Asserted if the object is an array.
`Linked`	1-bit	Asserted if the handle table entry is linked into the handle list.

Table 4.20: Header flags for memory management

Reachability_lvl: This flag indicates the reachability level of the object. It may hold one of the following five values: 0 (*unreachable*), 1 (*phantom reachable*), 2 (*weakly reachable*), 3 (*softly reachable*) and 4 (*strongly reachable*). If this flag is equal to 0 after the mark phase, the object is no longer referenced by the executing application and therefore can be garbage collected.

GC_Cycle_ID: This 2-bit flag stores the ID of the GC cycle in which the reachability level of the object is assigned. It is updated by the tracer during the mark phase. The reason for introducing this flag is that although the tracer marks live objects from both of the heap semi-spaces, the compactor compacts only one semi-space each time. A live object from the other semi-space could become unreachable in between the current and next GC cycles. Therefore, a timestamp is required to indicate whether the reachability level of an object is still valid. Otherwise, the reachability level of every live object from the other semi-space would have to be cleared additionally at the end of a GC cycle. For the tracer, a single bit should be enough to distinguish two consecutive cycles. However, a heap semi-space is compacted once every two GC cycles. Thus, the compactor needs a 2-bit cycle ID so that it can determine in which cycle the reachability level is actually assigned. If some object with an obsolete cycle ID is encountered during the compact phase, it can be garbage collected directly, without the need for checking its reachability level.

One_dim_prim_array: This flag is asserted by the allocator to facilitate tracing references. An one-dimensional primitive array does not contain any object reference and therefore can be ignored by the tracer in the mark phase.

Reference Tracing

The tracer is implemented based on the *iterative marking* algorithm proposed in [18], which is also adopted by the well-known *Boehm garbage collector* [17]. This algorithm utilizes a stack called *mark*

stack to buffer references to objects encountered in the mark phase. Each object is assigned a single-bit flag indicating whether the object has been marked. Thus, this flag is also called *mark bit*. If the mark bit of an object has been set, the reference to the object may not be pushed onto the stack again, which avoids recursive marking.

At the beginning of the mark phase, all mark bits must be cleared. Then, the references to the root objects are pushed onto the mark stack. During this process, the mark bits of the root objects are asserted. After that, references are traced in iterations. In each iteration, a reference is popped from the mark stack and the object that the reference points to is scanned. Only the references to unmarked objects that are held in the fields of the popped object are pushed onto the mark stack. The mark phase completes when the mark stack becomes empty. All unmarked objects are no longer reachable and therefore can be removed.

The algorithm described above was slightly modified due to the specific characteristics of the garbage collector built into the AMIDAR processor. The modified version differs from the original one in two ways. First, it uses both of the reachability level and the GC cycle ID rather than the mark bit to denote whether an object has already been encountered in the current mark phase. This eliminates the need for resetting all mark bits at the beginning of every GC cycle. Second, it also updates the reachability level of an object as the object is pushed onto the mark stack[5].

In the following, the reference tracing process executed in the mark phase is explained in detail. This process begins with the initialization of the GC stack by pushing the entire root set onto it, which is formally described in Algorithm 1 below.

Algorithm 1: GC stack initialization

```
input  : gcCycleID, gcStack, rootSet
output: void
1  gcCycleID = gcCycleID + 1;
2  reset(gcStack);
3  // push the entire root set onto the mark stack
4  foreach root in rootSet do
5      valid = getFlag(root, Valid);
6      rlvl = getFlag(root, Reachability_lvl);
7      gccid = getFlag(root, GC_Cycle_ID);
8      // valid check
9      if valid == 1 then
10         // duplicate check
11         if (rlvl == 0) || (rlvl != 0 && gccid != gcCycleID) then
12             setFlag(root, GC_Cycle_ID, gcCycleID);
13             setFlag(root, Reachability_lvl, 4);          // mark root as strongly reachable
14             push(gcStack, root);
15         end
16     end
17 end
```

[5] In the context of the AMIDAR processor, the GC stack described in Section 4.5.2 serves as the mark stack.

As mentioned in Section 4.2.5 and 4.4.3, the root set consists of the handle of the static field object (i.e. handle 1) as well as any object handles in the local variable array and operand stack of any stack frame. Therefore, the tracer first reads the object handles stored on the frame stack one by one via the dedicated interface between it and the frame stack.

Before an incoming handle is pushed onto the GC stack, the tracer performs two checks on it: a valid check (line 9) and a duplicate check (line 11). The valid check ensures that the handle does not belong to a WB or an immortal object. A WB or an immortal object (except the static field object) does not reference any dynamically created object that can be garbage collected. Thus, it is unnecessary to include such an object into the root set. To perform the valid check, the tracer loads the first header word of the object addressed by the handle from the handle table cache and buffers it into a local register. The high 16 bits of the word hold all flags associated with the object, while the low 16 bits hold the CTI of the class from which the object has been created. If flag `Valid` is not asserted, the check fails and the handle is simply skipped by the tracer. Note that the valid check also eliminates any null object reference (i.e. handle 0) from the root set implicitly because all flags of the 0-th handle table entry remain unset all the time.

Since an object may be referenced at different places in a program, its handle can be returned by the frame stack multiple times. The duplicate check is intended to avoid that a handle is pushed onto the GC stack more than once. To meet this goal, the tracer first checks the value of `Reachability_lvl` held in the buffered header word. If it is equal to 0, the object was created in between the previous and current GC cycles, i.e. it has never been marked since its creation. Thus, its handle can be pushed onto the GC stack safely. Otherwise, the tracer needs to additionally determine when the object's reachability level was assigned, in the current or previous GC cycle. For this purpose, it compares the value of `GC_Cycle_ID` of the object with the ID of the current GC cycle. If both values are the same, the handle has already been encountered at least once in the current mark phase and therefore should be abandoned this time, i.e. the check fails.

If the handle passes both of these checks, it is pushed onto the GC stack. The tracer marks the corresponding object as strongly reachable by assigning a constant value of 4 to `Reachability_lvl` in the buffered header word (line 13). At the same time, `GC_Cycle_ID` is also updated with the ID of the current GC cycle. After that, the buffered header word is written back to the handle table cache. This process repeats itself until the frame stack signals that all object handles have been transfered. Following that, the tracer pushes handle 1 onto the GC stack in addition without performing the checks above and marks the static field object as strongly reachable. At this moment, all objects whose handles can be found on the GC stack are strongly reachable.

After initializing the GC stack, the tracer iteratively traverses the graph of references starting from the root set and assigns each object encountered a reachability level. In most cases, if an object is referenced by another, the tracer simply passes on the reachability of the referencing object to the referenced one unless the referencing object is an instance of type **`Reference`**. In this case, the tracer additionally compares the reachability of the referencing object with the reachability associated with the class from which the referencing object has been created and assigns the referenced object the weaker one. For example, assume that an object O is referenced by an object of class **`SoftReference`**, namely O_{SR}. O is

marked as softly reachable, if O_{SR} is strongly reachable. However, if O_{SR} is weakly reachable, O is marked as weakly rather than softly reachable. Algorithm 2 describes the modified marking process formally.

Algorithm 2: Iterative marking

```
   input  : gcCycleID, gcStack
   output: void
 1 // trace out the graph of references from the root set
 2 while !isEmpty(gcStack) do
 3     parent = pop(gcStack);
 4     prlvl = getReachabilityLevel(parent);
 5     children = getReferenceFields(parent);
 6     foreach child in children do
 7         valid = getFlag(child, Valid);
 8         crlvl = getFlag(child, Reachability_lvl);
 9         gccid = getFlag(child, GC_Cycle_ID);
10         // valid check
11         if valid == 1 then
12             // duplicate check
13             if (crlvl == 0) || (crlvl != 0 && gccid != gcCycleID) then
14                 setFlag(child, GC_Cycle_ID, gcCycleID);
15                 setFlag(child, Reachability_lvl, prlvl);
16                 push(gcStack, child);
17             else if crlvl < prlvl then
18                 setFlag(child, Reachability_lvl, prlvl);
19                 push(gcStack, child);                  // push child onto the GC stack again
20         end
21     end
22 end
```

To realize the algorithm above, the tracer performs the following three steps repeatedly until the GC stack becomes empty:

1. The tracer pops the handle on the top of the GC stack. This handle and the object addressed by it are called the *parent handle* and the *parent object* respectively below. Through the parent handle, the tracer loads the first header word of the parent object into a local register. Then, the tracer determines the reachability level of the parent object, which should be passed on to the objects referenced by the parent object (line 4).

2. The tracer determines whether the parent object references other objects. If the parent object is a regular object, this information can be retrieved from the GC info module by providing the CTI held in the buffered header word. If the parent object is an array, the tracer checks flag `One_dim_prim_array` accordingly. If the parent object, whether a regular object or an array, does not reference any object, the tracer returns back to the first step, otherwise it goes to the next step.

3. The tracer pushes the handles of the objects referenced by the parent object onto the GC stack in sequence. If the parent object is a regular object, the offset of each of its reference fields is provided by the GC info module. If the parent object is an array, the offset of each of its elements

is generated by using an incremental index counter. Based on the parent handle and the offset of a reference field (or an array element), the tracer reads out the current value of the field (or the element) from the object cache. Then, it performs the valid check and the duplicate check on this value. According to the results of the checks, three different cases can occur:

- If the valid check fails, the tracer simply ignores the value and starts reading the next value from the object cache.
- If both of these checks succeed, the handle represented by the value is pushed onto the GC stack. The object addressed by the handle inherits the reachability from the parent object. Also, `GC_Cycle_ID` of the object is synchronized with the ID of the current GC cycle.
- If the valid check succeeds but the duplicate check does not, the handle represented by the value has already been encountered at least once. In this case, the tracer checks the reachability of the object addressed by the handle in addition. If the object has a weaker reachability than the parent object, the tracer replaces its reachability level with that of the parent object and pushes it onto the GC stack again. Otherwise the handle is abandoned.

After all reference fields (or array elements) of the parent object have been marked, the tracer goes back to the first step.

Although the handle of a marked object may be pushed onto the GC stack again, the modified iterative marking algorithm guarantees that there can be no duplicate handles on the GC stack due to the following facts. First, all objects whose handles are pushed onto the GC stack in iteration i must be assigned the same reachability, namely R_i. Second, the objects whose handles are pushed onto the GC stack in iteration $i+1$ cannot have a reachability stronger than R_i because their reachability is limited by that of their parent object. This has the consequence that the reachability levels of the objects whose handles are kept on the GC stack increase monotonically from the bottom to the top of the stack[6]. Therefore, if a marked object is encountered, which has a weaker reachability than its parent object, its handle cannot be buffered on the GC stack currently. Since the GC stack does not contain any duplicate handles, its size is set to the maximum number of handles allowed in order to avoid stack overflow.

Heap Compaction

After the mark phase, the resources occupied by unreachable objects need to be reclaimed and made available again to the executing application, which is the major task of the compactor. Besides that, the compactor also assists with handling special objects. From the viewpoint of the compactor, an object is considered special in one of the following two cases:

- It is no longer referenced by the application and has not been finalized yet.
- It is an instance of type **`Reference`** and has not been enqueued yet.

This subsection presents the process of reclaiming resources, while the next subsection explains how the compactor deals with special objects.

[6] A special case is that no instance of type **`Reference`** is used in the executing application, which causes that all live objects are strongly reachable.

The most fundamental operation that the compactor performs is the reallocation of objects inside the compaction space. For this purpose, it maintains two pointers in local registers, which are referred to as the *read pointer* and the *write pointer* respectively below. Using these pointers, the compactor can easily copy data from one memory location to another. At the beginning of the compact phase, both pointers are set to the base physical address of the current compaction space.

The compaction space may contain two different kinds of entities: objects and legacy memory gaps left by the allocator. They distinguish from each other by their first words. As described in Section 4.5.4, the first word of an object corresponds to an extra field holding the object's handle. This implies that the most significant bit of the word must be 0 because the highest handle bit may only be set for identifying a WB object which is, however, not allocated from the heap. In contrast, the first word of a memory gap is employed to keep the gap's size and is marked by setting its most significant bit. Therefore, each time after the read pointer has been assigned a new address, the compactor first reads out the word stored at this address from the main memory and then checks the highest bit of the word.

If the word belongs to a legacy memory gap, the compactor simply increases the read pointer by the gap's size that is included in the word. If the word represents the handle of an object, the entire header of the object is loaded from the handle table cache into local registers of the compactor. To determine the status of the object, the compactor checks the following two flags held in the buffered header: `Reachability_lvl` and `GC_Cycle_ID`. The object is reachable, if `Reachability_lvl` is greater than 0 and `GC_Cycle_ID` is equal to the ID of the current GC cycle, otherwise it is unreachable.

An object needs to be reallocated if either of the following conditions is met: 1. it is reachable and not locked (i.e. flag `Locked` is not set). 2. it is unreachable but has not been finalized yet (i.e. flag `Finalized` is not set). The former condition should be quite obvious, while the latter one could be a little counterintuitive. The only reason for reallocating an unreachable object is that its **`finalize`**-method has not executed, which could resurrect the object again. Therefore, an object may not be removed from the heap before it has been finalized. Currently, the compactor supports two different reallocation modes: the *incremental mode* and the *gap-based mode*. The former mode is activated by default and works in a straightforward way as presented below.

At the beginning of the compact phase, the read and write pointers are assigned the base physical address of the compaction space. As long as the read pointer points to a live object or an unreachable object that has not been finalized, both pointers are simply increased by the object's size. During this process, no object will be reallocated.

Once a legacy memory gap or a finalized unreachable object is encountered, only the read pointer is increased by the size of the gap or the object. In the latter case, the compactor also reclaims the object's handle, which is described at the end of this subsection. After the two pointers have become unequal, the compactor will copy every object that needs to be reallocated from the position addressed by the read pointer to the position addressed by the write pointer. Before the copy process, it first ensures that the object is not being accessed by the object cache. After the copy process, the physical address of the object is updated with the one held by the write pointer through the handle table cache. Then, both of the read and write pointers are increased by the object's size. Algorithm 3 describes the incremental reallocation formally.

Algorithm 3: Incremental reallocation

input : readPointer, writePointer, handle
output: void

```
1 while inUse(handle) do
2 |  // wait until current access completes
3 end
4 size = getObjSize(handle);
5 copy(readPointer, writePointer, size);
6 setObjAddr(handle, writePointer);
7 readPointer = readPointer + size;
8 writePointer = writePointer + size;
```

The compactor stays in the incremental mode until the read pointer reaches the end of the compaction space or points to a locked live object[7]. In both cases, a memory gap will be created whose size is equal to the difference between the read and write pointers. In the former case, the gap's forwarding pointer is set to 0, because this gap is already the last one in the current compaction space. Since the whole compaction space has been compacted, the header of the new gap is written to the address held by the write pointer. Then, the compactor writes 0 across the rest of the gap so that objects can be allocated from the gap directly. In the latter case, the gap's forwarding pointer is calculated by adding the read pointer and the size of the locked object together. Since further objects will be copied into the new gap, the gap's size will still vary. Due to this, the gap's size and forwarding pointer are buffered into local registers of the compactor to facilitate the following operations. Below, the gap addressed by the write pointer is referred to as the current gap.

The compactor reallocates objects in the gap-based mode as long as the addresses held by the read and write pointers are not included in the address range of the same gap. This means that there is at least one locked object in between both pointers. To determine when to switch back to the incremental mode, the compactor utilizes a third pointer besides the other two, which is called the *gap pointer*. As the compactor switches to the gap-based mode, the gap pointer is assigned the sum of the read pointer and the size of the locked object which causes this switch. Consequently, the gap pointer and the forwarding pointer of the current gap hold the same address at this moment. In the course of the gap-based reallocation, two different cases can happen:

- The read pointer reaches the end of the compaction space or another locked object. In this case, the compactor will check the size of the space between the read and gap pointers. If the size is larger than two words, the compactor will create a new gap at the position addressed by the gap pointer in the main memory directly. However, this gap will not be initialized (i.e. zeroed). The forwarding pointer of the gap is assigned the sum of the read pointer and the size of the locked object. With this sum, the gap pointer is updated as well, despite whether a new gap can be created. This indicates that the gap pointer always holds the first address following the latest locked object encountered.

[7] A locked live object is simply referred to as a locked object in the following description because the `Locked` flag of an unreachable object is ignored by the compactor.

- The current gap does not contain enough space into which the next object that needs to be reallocated can fit. In this case, the buffered header of the current gap will be written into the memory location addressed by the write pointer, if the gap's size is larger than two words. Also, the write pointer is assigned the address held by the forwarding pointer of the current gap. If the updated write pointer and the gap pointer are equal, the compactor switches back to the incremental mode. Otherwise, the header of the new current gap is loaded from the main memory into local registers.

As Algorithm 4 illustrates, before the compactor reallocates an object in the gap-based mode, it must first check whether the current gap is large enough to keep this object. If the object can fit into the remaining space of the gap, it is copied to the position addressed by the write pointer. The copy process is executed in exactly the same way as in the incremental mode (line 3) except that the size of the current gap needs to be updated additionally (line 4). If the current gap is too small to store the object, the compactor writes the header of the current gap to the address held by the write pointer (line 7) and then skips to the next gap (line 8). If the new current gap is already the last gap, the compactor switches to the incremental mode (line 10) and then reallocates the object (line 11). Otherwise, it loads the header of the new current gap into local registers and performs the gap-based reallocation recursively (line 16) until the object is reallocated successfully. Like the allocator, once the compactor leaves a gap, it will not attempt to reallocate any other object into the gap in the current GC cycle. After the read pointer has reached the end of the compaction space, the compactor zeros all gaps generated at a time.

Algorithm 4: Gap-based reallocation

```
   input  : readPointer, writePointer, gapPointer, handle, gapSize, gapFP
   output: void
 1 size = getObjSize(handle);
 2 if gapSize >= size then                                  // current gap is large enough
 3 |   incrementalRealloc(readPointer, writePointer, handle);
 4 |   gapSize = gapSize - size;
 5 end
 6 else                                                                  // skip to next gap
 7 |   createGapHeader (writePointer, gapSize, gapFP);
 8 |   writePointer = gapFP;
 9 |   if writePointer == gapPointer then
10 |   |   incrModeOn();                          // switch back to the incremental mode
11 |   |   incrementalRealloc(readPointer, writePointer, handle);
12 |   end
13 |   else
14 |   |   gapSize = getGapSize(writePointer);
15 |   |   gapFP = getGapForwardingPointer(writePointer);
16 |   |   gapBasedRealloc(readPointer, writePointer, gapPointer, handle, gapSize, gapFP);
17 |   end
18 end
```

To better illustrate the process of reallocating objects, a simple example is presented below, which is based on the example shown in Figure 4.20. Assume that semi-space B needs to be garbage collected immediately after semi-space A has been compacted. It contains a total of five live objects, namely object

3, 7, 8, 9 and 10, where object 7 is locked. In addition to them, there is also an unreachable object that has not been finalized, namely object 6. As Figure 4.21 demonstrates, the read and write pointers are set to the base address of the semi-space at the beginning of the compact phase (snapshot 0).

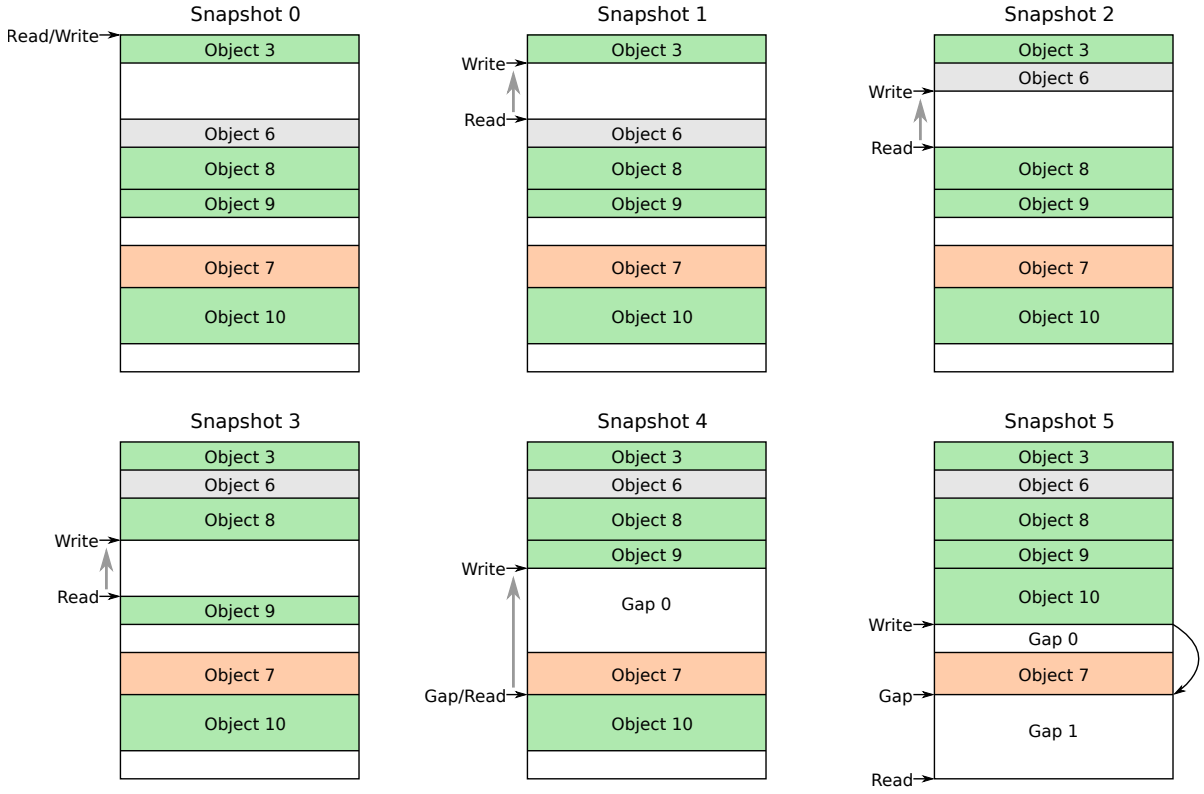


Figure 4.21: Snapshots of the compaction space

Object 6, 8 and 9 are reallocated in the incremental mode (snapshot 1-3). Due to the occurrence of object 7, the compactor switches to the gap-based mode and creates gap 0. Also, the gap pointer and the forwarding pointer of gap 0 are assigned the first address following behind object 7 (i.e. the address of object 10). After object 10 has been copied into gap 0, the read pointer reaches the end of semi-space B. This causes that the header of gap 0 is written to the memory address held by the write pointer. Additionally, a new gap, namely gap 1, is created at the position addressed by the gap pointer. Finally, the compactor zeros both gaps to make them available to the allocator.

Besides the object reallocation, the compactor also establishes a handle list in the compact phase. Each time the read pointer points to an unreachable object that has been finalized, the compactor clears the current flags and values held in the object's header and marks the header as linked through the handle table cache. If the handle list is still empty, this header becomes its first entry. Otherwise, the object's header is attached to the existing list by writing the object's handle into the previous list entry.

Handling Special Objects

Two kinds of objects need to be specially handled in the compact phase: unreachable objects that have not been finalized and reachable objects of type **`Reference`** that have not been enqueued. If an object of the former kind is encountered, the compactor checks whether the object has a nonempty **`finalize`**-method by providing the CTI of the object's class to the GC info module. If this is the case, the

object is marked as finalized and its handle is pushed onto the GC stack. Otherwise, the object's handle is reclaimed by the compactor directly.

If an object of the latter kind is encountered, the compactor needs to determine whether the referent of the object has the corresponding reachability level as that of the class from which the object has been created (e.g. whether the referent is softly reachable, if the object is an instance of class **SoftReference**). For this purpose, the compactor reads out the handle of the referent from the object cache. Then, the referent's reachability level is retrieved from the handle table cache and compared with that of the object's class. Upon a match, the compactor treats the object as follows:

- If the object is an instance of class **SoftReference** or **WeakReference**, the compactor overwrites the handle of its referent with a value of 0 (i.e. null) via the object cache. Also, the compactor asserts the highest bit of the object's handle and pushes the altered handle onto the GC stack.
- If the object is an instance of class **PhantomReference**, an extra check is performed on its referent, where three different cases can happen:
  - If the referent has not been finalized and has a nonempty **finalize**-method, it is marked as finalized and its handle is pushed onto the GC stack.
  - If the referent has not been finalized but has an empty **finalize**-method, it is marked as finalized and the object is marked as enqueued. Additionally, the object's handle is altered by asserting its highest bit and then is pushed onto the GC stack.
  - If the referent has been finalized, the object is marked as enqueued. After the highest bit of the object's handle is asserted, the altered handle is pushed onto the GC stack.

As mentioned above, the garbage collector contains a software component, namely the GC thread, which is employed to finalize unreachable objects and to enqueue instances of type **Reference**. It has been implemented as the interrupt service thread of the heap manager and is assigned the lowest priority to interfere as little as possible with the execution of the application. The interrupt handling model of the AMIDAR processor is described in Section 4.6.5 in detail.

Most of the time, the GC thread suspends itself by invoking the **wait**-method. If the GC stack is not empty at the end of the compact phase, the heap manager will wake up the GC thread by sending an interrupt request to the thread scheduler. The awakened GC thread reads out the entries on the GC stack in sequence via the WB interface. According to the highest bit of a stack entry, the GC thread can determine which method should be invoked, **finalize()** of class **Object** or **enqueue()** of class **Reference**. Note that values returned by the WB interface are all integers. Exploiting the **intToRef**-method of class **de.amidar.AmidarSystem**, they can be converted to handles. If a value corresponds to the handle of an object of type **Reference**, its highest bit needs to be cleared before the conversion. Once the GC stack becomes empty, the WB interface returns a value of 0 to notify the GC thread. As a result, the GC thread starts waiting again. If a new GC cycle is triggered before the GC stack becomes empty, the tracer will clear the `Enqueued` and `Finalized` flags of all objects whose handles still remain on the GC stack and then reset the GC stack.

As one might think, an object of type **Reference** could be simply identified by using the **instanceof** operator of Java. However, a class derived from one of the three subclasses of **Reference** might also

define a nonempty **finalize**-method. If the **instanceof** operator was adopted, the GC thread could not determine which method should be invoked on the object of such a class.

4.5.6 Wishbone Object Access

Inside the AMIDAR processor, a WB object is just treated as a regular object. This means that any of its fields needs to be accessed by using its handle and the field's offset. Once the object cache receives both of these values, it will identify the WB object due to the asserted highest handle bit and consequently generates a 32-bit WB address as shown in Figure 4.22. Then, this access is redirected via the AMIDAR infrastructure described in Section 4.1.3 to the corresponding peripheral.

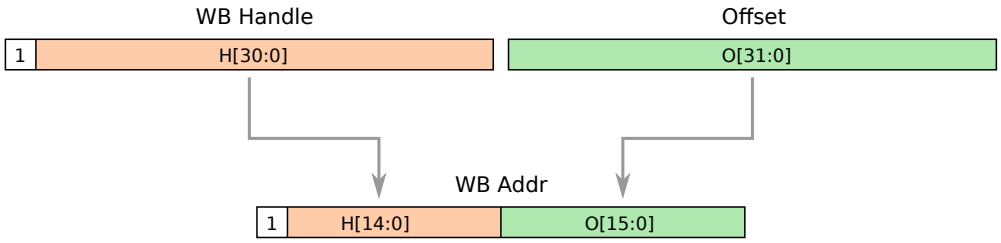


Figure 4.22: Generation of WB address

The addressing scheme illustrated above allows up to 2^{15} peripherals to be connected to an AMIDAR-based SoC at the same time, each of which may have a maximum of 2^{16} registers. This should be sufficient to cover all real-world scenarios in the field of embedded systems.

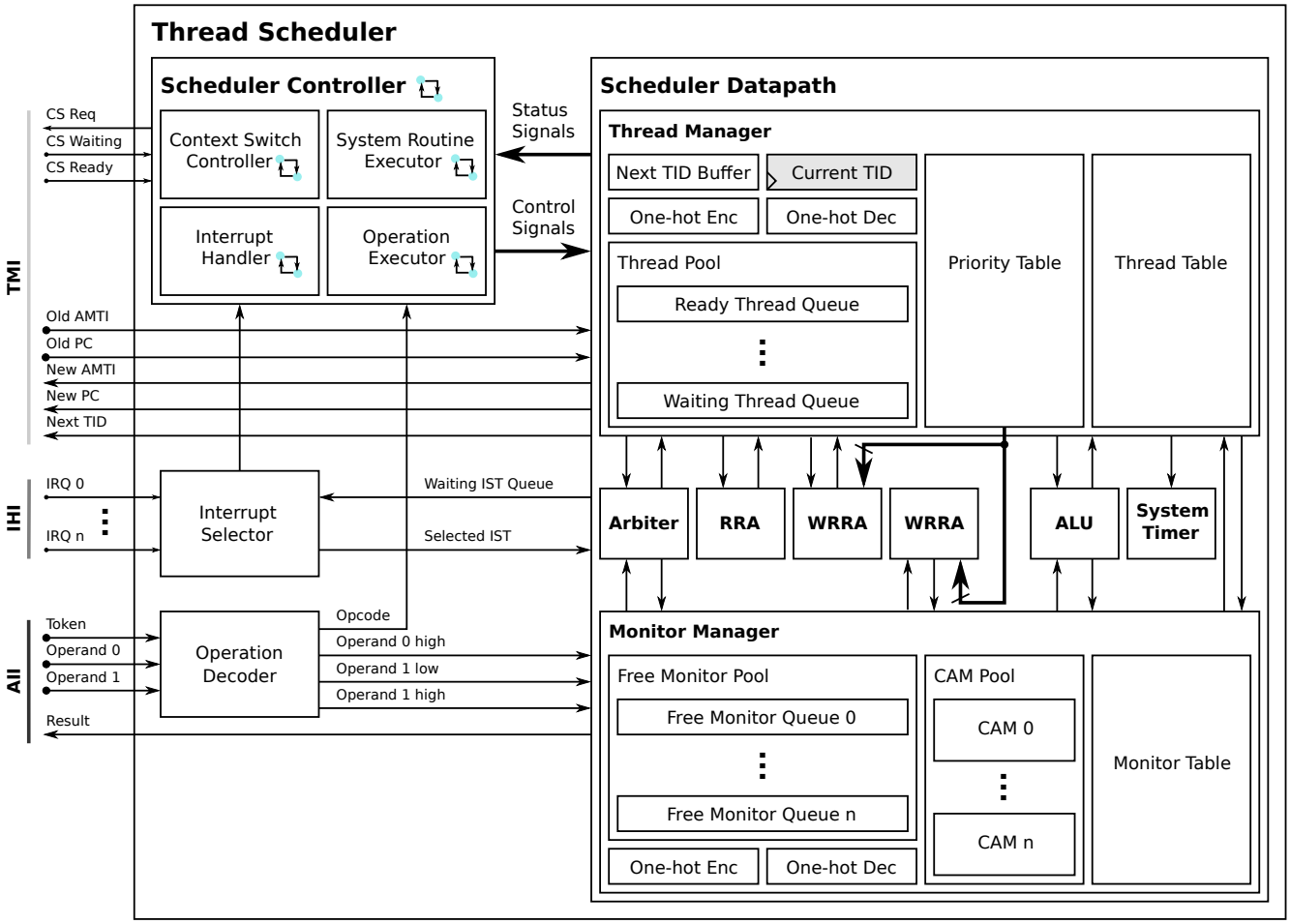
4.6 Thread Scheduler

This section introduces the thread scheduler of the AMIDAR processor. It supports the most essential thread- and synchronization-specific operations in hardware directly. Also, it provides an elegant interrupt handling mechanism that has been seamlessly integrated into the thread scheduling framework, allowing an interrupt service routine to be implemented in terms of a regular Java thread.

Figure 4.23 illustrates the structure of the scheduler. In the following, we first describe the most relevant datapath components shown on the right side of the figure. Then, we give detailed information about the hardware-software interface that allows for efficient interactions between the scheduler and a Java program. After that, Section 4.6.3 introduces how a thread is managed inside the scheduler, from its creation to its termination. Section 4.6.4 explains the implementation of the Java monitor construct and the last section presents the interrupt handling mechanism.

4.6.1 Datapath Components

The datapath components of the scheduler can be classified into two groups: key and utility components. The key components determine the primary behavior of the scheduler and include the arbiter, *round-robin arbiter* (RRA), *weighed round-robin arbiter* (WRRA), thread queue and priority table. The PQ architecture used in the scheduler is formed by combining the latter three key components together. All of the key components are discussed in detail in the following. The utility components are those that facilitate performing general operations such as converting formats or buffering data. For example, the thread/monitor tables and one-hot encoder/decoder belong to this group. We describe such components only briefly below.



TMI: *Token Machine Interface* **IHI:** *Interrupt Handling Interface* **AII:** *AMIDAR Infrastructure Interface*
RRA: *Round-Robin Arbiter* **WRRA:** *Weighted Round-Robin Arbiter*

Figure 4.23: Thread scheduler overview

Arbiter

This simple circuit selects the leftmost or rightmost nonzero bit (i.e. the first **1** bit on the left or right side) of a given binary value[8] and yields a binary value of the same size in which all bits are set to 0 except the selected bit. For example, given a binary value $r = 0101\_0100_2$, an 8-bit arbiter returns the one-hot value $g = 0100\_0000_2$ back.

Figure 4.24 illustrates a naive implementation of a n-bit arbiter consisting of a chain of n identical bit-cells[9]. Each bit-cell, e.g. the m-th, calculates a result bit $g[m]$ and a carry bit $c[m]$ respectively, using the corresponding input bit $r[m]$ and the carry bit $c[m+1]$ of the left neighbor bit-cell as follows:

$$\begin{aligned} g[m] &= c[m+1] \ \& \ r[m] \\ c[m] &= c[m+1] \ \& \ \sim r[m] \end{aligned} \tag{4}$$

[8] Note that we consider only the leftmost nonzero bit in the following description.
[9] The IEC symbols are used to represent different logic gates in this section.

If $r[m]$ is equal to 1, $c[m]$ is pulled down to 0, which causes that the rest of the carry bits from $c[m]$ to $c[1]$ become 0. Consequently, the result bits between $g[m-1]$ and $g[0]$ are masked to 0, despite the input bits.

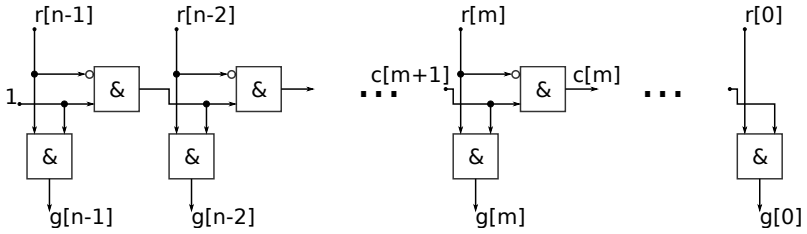


Figure 4.24: Basic arbiter architecture

The major problem of the basic arbiter architecture is that the length of the carry chain increases proportionally to the width of the input value, limiting its scalability, especially, when clock frequency is taken into account. Therefore, we propose an extended arbiter architecture with lookahead, upon which a 64-bit arbiter instance is constructed and shown in Figure 4.25, for example.

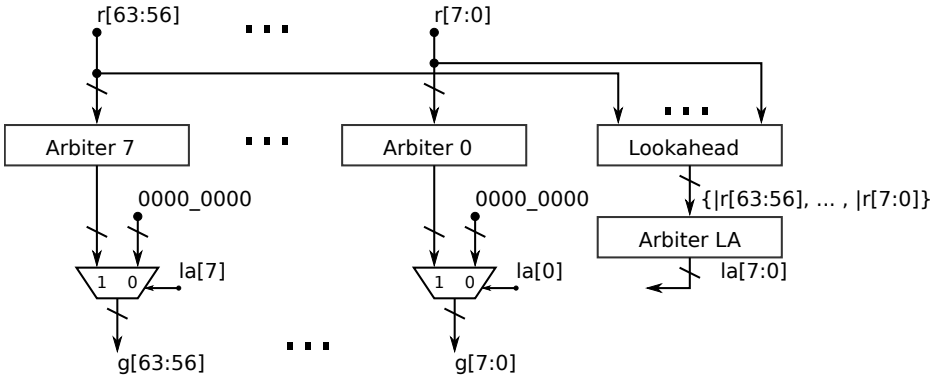


Figure 4.25: Extended arbiter architecture

The key idea of this architecture is to partition the input value into several bit-vectors and input them into multiple small basic arbiters in parallel. Only the leftmost nonzero result among all results of these arbiters remains unchanged in the final output value, whereas other results are replaced with 0, using multiplexers. The value of the select signal of each multiplexer is calculated as follows. First, a lookahead bit for each bit-vector is generated by performing the or-reduction on the bit-vector. This can be efficiently realized in hardware, exploiting a binary tree of OR gates. A nonzero lookahead bit indicates that the corresponding bit-vector contains at least a nonzero bit. Then, all lookahead bits are concatenated in the order of the bit-vectors from left to right and fed into another basic arbiter. Each bit of the result of the arbiter is connected to the select signal of the corresponding multiplexer. Table 4.21 provides the resource usages and maximum clock frequencies[10] of the 64-bit arbiters with and without lookahead, which are measured using Xilinx Vivado v2017.2 on an Artix-7 FPGA [75]. As the table illustrates, the extended arbiter improves the clock frequency greatly at the slight expense of hardware overhead.

[10] Both arbiters are combinational circuits. For the purpose of timing analysis, they are first turned into sequential circuits by inserting I/O registers and clock signals. The maximum clock frequencies of both sequential circuits, which can be reached without causing any negative slack, are considered as the maximum clock frequencies of the arbiters.

Architecture	LUT	Max. frequency
Basic arbiter	99	175MHz
Extended arbiter	106	250MHz
Δ	+7%	+42%

Table 4.21: Comparison of 64-bit arbiters with and without lookahead

The arbiter circuit is utilized in a broad variety of situations. Traditionally, it is used to handle requests from multiple users/devices that compete for a single common resource like a bus. From the standpoint of scheduling, it can also be considered as a bitmap scheduler mentioned in Section 2.3.4, which assigns each request a unique priority in descending order from left to right. Hence, the FIFO-PQ architecture described in Section 2.4 may adopt it to perform the priority-based inter-queue scheduling. In the scheduler of the AMIDAR processor, it is employed to construct more sophisticated circuits like RRA and to manage free entries in the thread and monitor tables.

Round-Robin Arbiter

The RRA circuit grants access permission for a common resource to multiple users/devices in a round-robin manner, ensuring the fairness among them. It can be easily implemented using the arbiter circuit described above and a masking logic, as shown in Figure 4.26. The arbiter built into the RRA circuit contains an output register holding its result.

The primary task of the masking logic is to filter out the requests that have not been granted in the current arbitration round yet, exploiting a mask register. Also, it needs to update the value held in the mask register according to the granted request. Assume that r, m, m' and rm represent the input requests, the current and updated values of the mask register as well as the masked requests in n-bit binary form respectively, while g represents the granted request in n-bit one-hot form, the masking logic can be formally defined as follows:

$$rm = \begin{cases} r, & \text{if } r \,\&\, m = 0 \\ r \,\&\, m, & \text{otherwise} \end{cases} \tag{5}$$

$$m' = \begin{cases} g \oplus r, & \text{if } r \,\&\, m = 0 \\ g \oplus rm, & \text{otherwise} \end{cases} \tag{6}$$

In the mask register, each bit that is set to 1 represents a request that has not been granted yet. Once all remaining requests have been granted or canceled (i.e. $r \,\&\, m = 0$), the current round is completed. Note that there is no intermediate register between the masking circuit and the arbiter. Thus, the RRA circuit takes only a single clock cycle to select the next request to grant. However, since the result of the arbiter is registered in an output buffer, as mentioned above, the mask register is actually updated at the rising edge of the next clock cycle.

The RRA circuit has 3 handshaking signals, namely en, $valid$ and $idle$, which are described below:

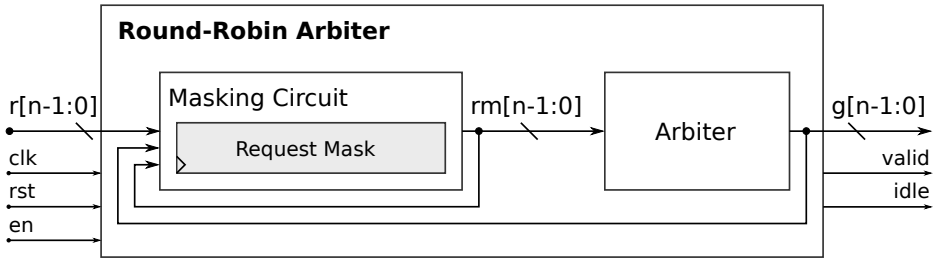


Figure 4.26: Round-robin arbiter

- en: When active, the RRA circuit starts selecting the next request to grant and outputs the selected request in one-hot form one clock cycle later.
- $valid$: This signal is asserted for one clock cycle to indicate that the output port g contains a valid result.
- $idle$: This signal is asserted as long as $m = 0$, i.e. there is currently no started arbitration round. Note that it is asserted one cycle later than the $valid$ signal due to the way how the mask register is updated.

To better illustrate the functionality of the RRA circuit, the following example simulates 2 sample arbitration rounds of a 4-bit RRA. Before the first sample round begins, there are 3 requests to be handled. After the first round has been started, another two requests arrive. However, both new requests cannot be granted until the second round, since they are not held in the current mask. This implies that a total of 3 arbitrations need to be performed in the first round. At the beginning of the second sample round, there are also 3 requests. However, one of them is canceled after the first arbitration. Therefore, the second round includes only 2 arbitrations. The minimum interval between two arbitrations is one clock cycle, and there is no limit on the maximum interval. Table 4.22 shows the two sample rounds in more detail. Both new requests in the first round are marked in green, while the canceled one in the second round is marked in red. Figure 4.27 demonstrates the timing of important signals of the 4-bit RRA according to the second sample round.

Round	Arbitration	r	$m_{current}$	r & m	rm	g	m_{next}
1	1	1011_2	0000_2	0000_2	1011_2	1000_2	0011_2
	2	0111_2	0011_2	0011_2	0011_2	0010_2	0001_2
	3	1101_2	0001_2	0001_2	0001_2	0001_2	0000_2
2	1	1101_2	0000_2	0000_2	1101_2	1000_2	0101_2
	2	0001_2	0101_2	0001_2	0001_2	0001_2	0000_2

Table 4.22: Two sample arbitration rounds of a 4-bit RRA

The RRA circuit should be used in circumstances where fairness is favored, to ensure that every request can be granted within a predictable delay. The scheduler of the AMIDAR processors exploits it to handle interrupts and check the timeout values of sleeping and waiting threads. It is also one of the key components of the WRRA circuit described below.

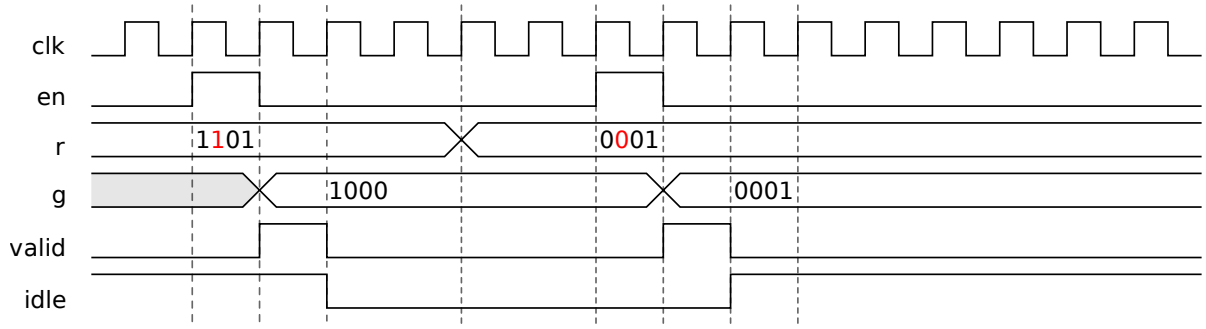


Figure 4.27: Timing diagram for the second RRA sample round

Weighted Round-Robin Arbiter

In contrast to the RRA circuit, the WRRA circuit considers additionally the priorities of requests during arbitrations. It guarantees that only after all requests with the same priority have been granted, the requests with the next lower priority may be handled. To meet this goal, the WRRA circuit needs a masking logic that sorts requests according to their priorities, which can be realized in different ways. One possible solution would be the implementation of a sorting algorithm that checks the priorities of valid requests sequentially. Such a WRRA needs a single priority input port (or two, when exploiting the dual ports of BRAM) and an FSM controlling the sorting process. The advantage of this solution is that the priorities of requests can be held in BRAMs, reducing the usage of on-chip registers. The drawback is that the sorting time varies depending on the number of requests. Since the scheduler of the AMIDAR processor aims to provide a constant scheduling time, we decided to use an architecture that accesses to all priorities simultaneously, as shown in Figure 4.28.

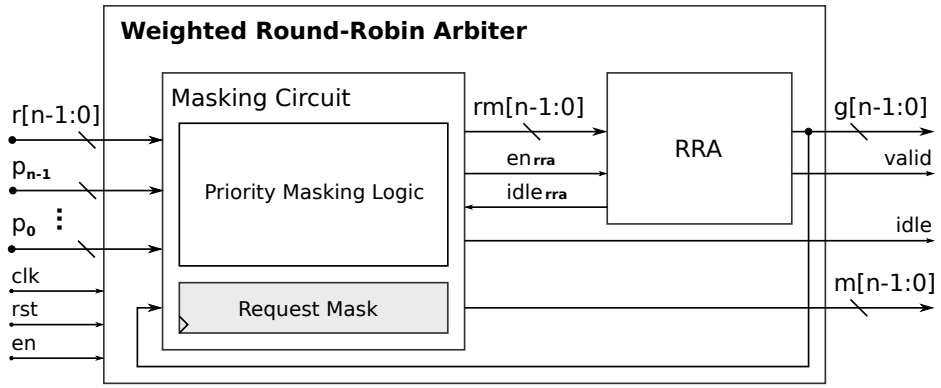


Figure 4.28: Weighted round-robin arbiter

The masking circuit has two major tasks. The first is to extract the requests with the currently highest priority and feed them to an internal RRA. The second is to remove granted requests from the mask register, allowing requests with lower priorities also to be handled. Figure 4.29 shows the logic built into the masking circuit, which carries out the former task. First, all input priorities are masked according to the request vector and the mask register to remove the priorities of invalid requests and

requests that have been granted in the current round. The following equation describes this process formally:

$$pm_i = \begin{cases} 0, & \text{if } r[i] \,\&\, m[i] = 0 \\ p_i, & \text{otherwise} \end{cases} \tag{7}$$

Then, the highest priority of all masked priorities is found out, exploiting a binary tree of comparators (BTC). At last, the requests with the highest priority can be extracted through comparing the highest priority with each of the masked priorities as follows:

$$rm[i] = \begin{cases} 1_2, & \text{if } p_i = p_{highest} \\ 0_2, & \text{otherwise} \end{cases} \tag{8}$$

The masked requests are buffered in an intermediate register in order to break the long data path between the input ports and the output register of the internal RRA. Also, the BTC can be pipelined to meet a given timing requirement. For example, an extra pipeline stage is inserted into the BTC for the 64-bit version of the WRRA circuit, which makes the masking process take two clock cycles totally. Note that the masking process is triggered only if the $idle$ signal of the internal RRA is asserted.

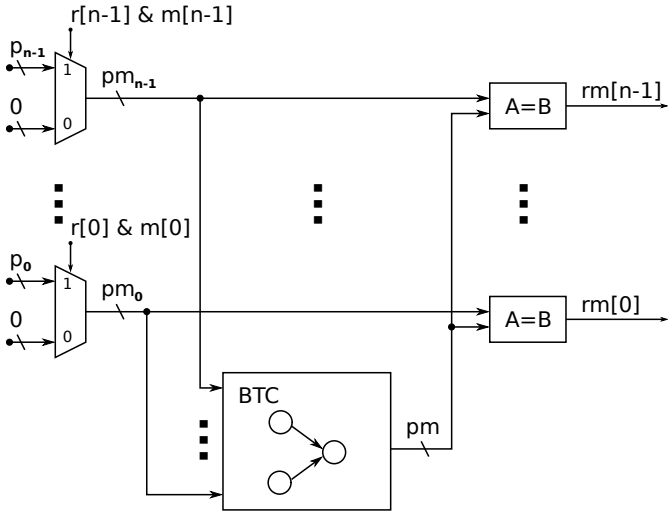


Figure 4.29: Priority masking logic of the WRRA circuit

The internal RRA arbitrates the masked requests one by one. Each time a request is granted, the masking circuit removes it from the mask register accordingly. Once all masked requests have been granted, the $idle$ signal of the RRA becomes active, signaling the masking circuit that the requests with the next lower priority may be extracted. Formally, the process for updating the mask register can be described as such:

$$m = \begin{cases} r, & \text{if } \; m = 0 \;\text{ and }\; en_{WRRA} = 1_2 \\ m \;\oplus\; g, & \text{else if } \; valid = 1_2 \end{cases} \tag{9}$$

Note that the WRRA circuit provides an additional port which outputs the current value of the internal mask register continuously. This port is employed for the purpose of queue switch.

Since the masking process costs extra clock cycles, the timing of the 3 handshaking signals of the WRRA circuit, namely en, $valid$ and $idle$, differs from that of the corresponding signals of the RRA circuit. The differences are discussed briefly below. Assume that a WRRA is just in the idle state before the en signal is set. After en becomes active, the masking circuit first updates the mask register according to Equation 9 above, i.e. assigns r to m. Then, it uses two clock cycles to mask the incoming requests[11] due to the asserted $idle$ signal of the RRA. After the requests with the currently highest priority have been extracted, the masking circuit asserts the en signal for the RRA. Consequently, the RRA outputs the selected request in one-hot form one clock cycle later after its en signal has be set. Therefore, it takes a total of 4 clock cycles to perform this single arbitration. However, after the first arbitration, the masking circuit only needs to forward the input en signal to the internal RRA directly. As a result, each of the following arbitrations needs only a single clock cycle until all the rest of the masked requests have been granted. Then, the RRA asserts its $idle$ signal, causing the masking circuit to select the requests with the next lower priority. In this way, the WRRA grants all requests in multiple sub-rounds. The first arbitration of each sub-round costs 4 clock cycles, whereas the other arbitrations of the same sub-round only one. The $idle$ signal of the WRRA is asserted at the end of the last sub-round, after the mask register has been updated and does not contain any nonzero bit anymore. The following example simulates two sample arbitration rounds of a 4-bit WRRA, illustrating the functionality of the WRRA circuit more clearly.

At the beginning of the first round, the 4-bit WRRA is idle. There are 2 valid requests that have the priority p. This indicates that the first round includes a single sub-round with two arbitrations. After the first arbitration, a third request with the priority $p+1$ arrives, which, however, cannot be handled in the first sample round. In the second sample round, the 3 requests remain unchanged and a fourth request with the priority $p+1$ arrives. These four requests are granted in 2 sub-rounds, one for each of both priorities. Table 4.23 shows detailed information about the two sample rounds. To better distinguish between different requests, we mark a request with the priority $p+1$ in green. Based on the second sample round, Figure 4.30 also shows the timing of important signals, including both interface signals of the WRRA and several internal signals between the masking circuit and the RRA.

Round	Arbitration	r	rm	g
1	1	0110_2	0110_2	0100_2
	2	0111_2	0110_2	0010_2
2	1	1111_2	1001_2	1000_2
	2	1111_2	1001_2	0001_2
	3	1111_2	0110_2	0100_2
	4	1111_2	0110_2	0010_2

Table 4.23: Two sample arbitration rounds of a 4-bit WRRA

Unlike the arbiter and RRA circuits described above, which are intended for general use, the WRRA circuit is used solely for the purpose of thread scheduling. According to this restriction, one impor-

[11] Note that this time can be varied depending on the specific timing requirement and the maximum number of requests. We use two in this section only for ease of discussion.

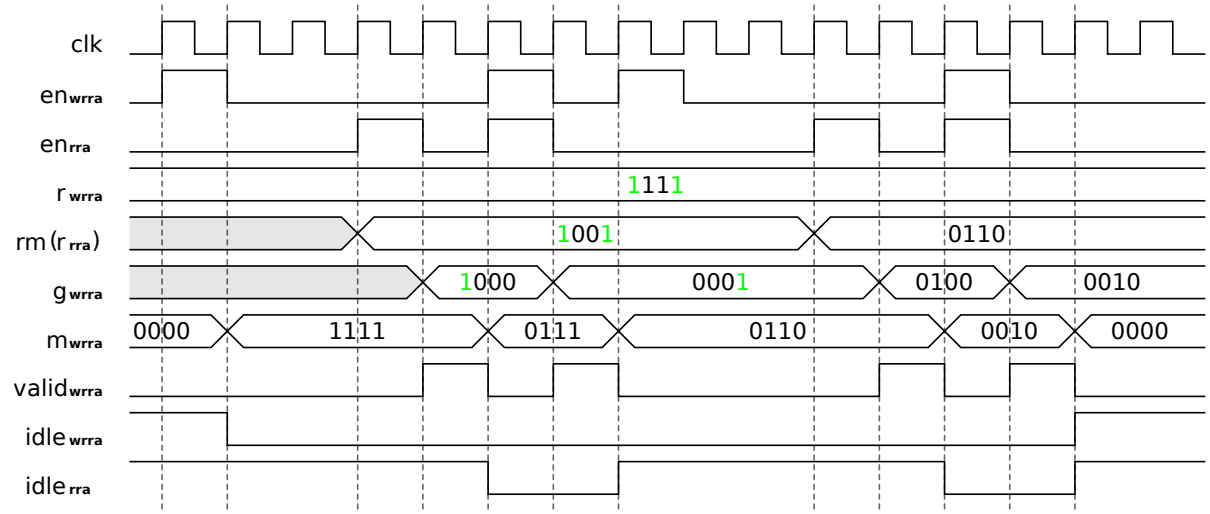


Figure 4.30: Timing diagram for the second WRRA sample round

tant design decision which has been made is that the WRRA circuit does not handle canceled requests autonomously like the RRA circuit. This has the consequence that the masked requests remain frozen throughout an entire arbitration sub-round, simplifying the implementation of the masking circuit. The reason for this design decision is given in the following sections, as we discuss the implementation of the Java thread and synchronization model.

Another key thing to note about the WRRA circuit is that it grants requests with the same priority according to their spatial positions in the request vector rather than their arriving order. This means that the arbitration result of the WRRA circuit cannot reflect the FIFO order of incoming requests generally. As the example above illustrates, the request with the priority $p+1$ that is associated with the lowest bit in the request vector arrives already in the first sample round. The other request with the same priority, i.e. the one that corresponds to the highest bit in the request vector, arrives at the beginning of the second sample round. However, the arbitration result shows that the latter request is granted before the former one, violating the arriving order of both requests. As mentioned above, the WRRA circuit is adopted for scheduling threads dedicatedly. Therefore, we discuss this issue solely in the context of thread scheduling and consider a request vector of the WRRA circuit as a thread queue, in which every valid request corresponds to a thread that needs to be scheduled.

Traditionally, the workload of an embedded system is represented as a task set containing both periodic and aperiodic tasks [16, 23, 37, 62, 105]. A thread executing a periodic task is a regular user thread, whereas a thread executing an aperiodic task typically corresponds to an IST. Since ISTs are scheduled using the RRA circuit, we only discuss regular user threads below, without consideration of thread synchronization and self-blocking (i.e. the ready thread queue stays in a steady state after the startup-phase). In general, a user thread should be started in the startup phase of an application. Then, it executes its task periodically until the application terminates. To allow the thread to be scheduled by the WRRA circuit, we need to assert some bit in the ready thread queue to represent the thread. If each newly started thread can always obtain the currently leftmost unused bit (i.e. the first *zero* bit on the left side of the ready thread queue), the spatial order of all user threads, when counted from left to right, coincides their starting order exactly. As a result, all user threads are scheduled in their starting order periodically throughout the whole lifetime of the application. The scheduler of the AMIDAR processor

utilizes a simple approach to meet this goal, which is explained in Section 4.6.3. Even if there was no such a mechanism, the influence of this issue should be negligible for most applications in the field of embedded systems. This is because the scheduling rate of a thread is more important than its scheduling order. The scheduling rate corresponds to the frequency at which a thread is scheduled in a given time internal. It determines the fairness among threads and should remain monotonic, which is therefore the primary aim of the well-known RM scheduling algorithm [67]. In contrast to the scheduling rate, the scheduling order affects only the time point at which a thread is assigned a time-slice to run for the first time. A key point is that the WRRA circuit provides each user thread a fixed scheduling rate, depending on the number of user threads and their priorities.

Thread Queue

A thread queue of size n is basically just a n-bit register, with the addition of the logic that supports the enqueue/dequeue operations, as shown in Figure 4.31. The port TQ_{binary} outputs the current thread queue in binary form. Each bit that is set in the register represents a valid thread in the queue. The status signal $empty$ is asserted if no bit in the register is equal to 1, i.e. the queue does not contain any valid thread. Through the input port $TID_{one\text{-}hot}$, a thread ID in one-hot form can be added to or removed from the queue by performing an OR- or XOR- operation on the given ID and the current value of the queue register. If both en and $dequeue$ become active, the dequeue operation is performed; otherwise, if only en is set, the enqueue operation is performed. The advantage of this queue architecture is that a thread can be saved in the queue by using a single bit. Its weakness is that a thread ID needs to be transformed between the one-hot and binary formats continuously. However, these transformations can be executed in hardware efficiently. Note that this circuit is also used for managing free monitors in the monitor manager.

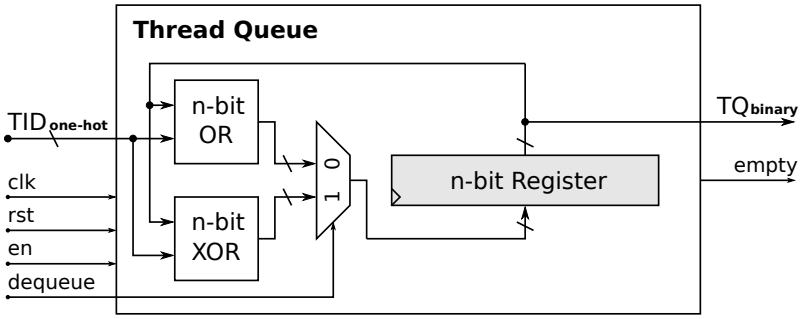


Figure 4.31: Thread queue

Priority Table

The priority table shown in Figure 4.32 is a simple register file that holds the priorities of all threads. In the standard Java specification, a total of 10 priority levels are defined, which means that each entry in the register file needs 4 bits to save a single priority value. The priority table has two access interfaces: the *random access interface* (RAI) and *parallel access interface* (PAI). The RAI supports random access to the priority of an arbitrary thread through the ID of the thread, which allows the priority to be easily changed at runtime. The port a of the RAI allows for both read and write operations, whereas the port b

only allows for the read operation. The PAI outputs all priorities in parallel, which are connected directly to the priority ports of a WRRA. As Figure 4.23 illustrates, this table can be shared by multiple WRRAs.

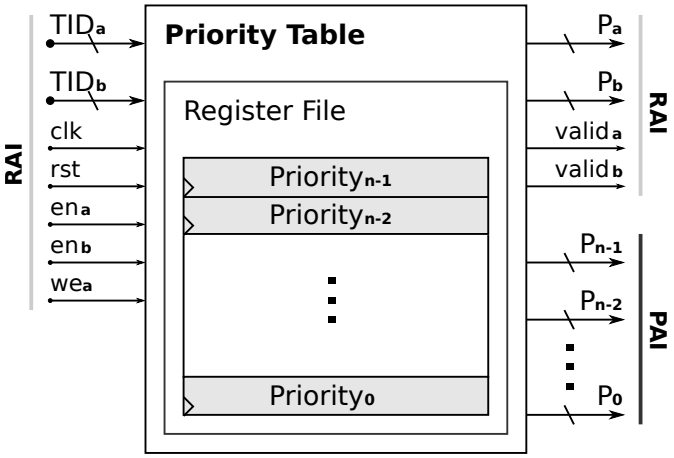


Figure 4.32: Priority table

WRRA-PQ

Figure 4.33 demonstrates the PQ architecture used in the thread scheduler of the AMIDAR processor, which is just a combination of the WRRA, thread queue and priority table circuits described above. It is a logical concept and not implemented as a dedicated hardware module. This loose construction form allows different PQ architectures to share common parts like priority table, as shown in Figure 4.23. In the rest of this thesis, we refer to this PQ architecture as WRRA-PQ.

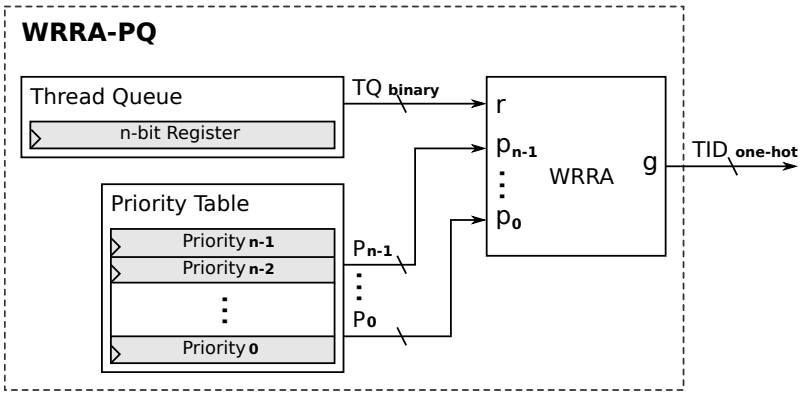


Figure 4.33: WRRA-PQ

The WRRA-PQ architecture provides the following key properties that are required for the implementation of a general-purpose thread scheduler:

- Support for the MLQ algorithm.
- Support for the RR inter-queue scheduling.
- Support for efficiently changing thread priorities at runtime.
- Support for being shared by multiple thread queues.

We discuss briefly how these properties are realized by using the three simple components of the WRRA-PQ architecture. The former two properties are provided by the WRRA circuit. As explained above, the WRRA circuit schedules all in-queue threads in multiple sub-rounds. These sub-rounds are performed in decreasing order of priority. Each sub-round can be considered as an intra-scheduling process at a single priority level. This fits the standard definition of the MLQ algorithm exactly. After the threads with the lowest-priority have been scheduled, the internal mask register is reset, allowing the whole scheduling process to be started from the highest priority again. This causes the inter-queue scheduling to be performed in a RR manner. Just like the SR- and SA-PQ architectures, the WRRA-PQ architecture uses a single physical queue to hold multiple logical queues.

The priority of a thread can be easily changed by writing the new value into the priority table through its RAI. There are two possible cases in which the priority of a thread can be changed when executing a Java program on the AMIDAR processor. One case is the invocation of the **`setPriority`**-method on a thread object and the other is the occurrence of a priority inheritance. According to the time point at which the priority of a thread is altered, there can be three different consequences. If the thread has already been scheduled in the current scheduling round, its new priority does not affect the scheduling result anymore until the next round. If the sub-round that corresponds to the old priority of the thread has been started and the thread has not been scheduled in this sub-round yet, the new priority of the thread will not change its scheduling order, but will change the length of its time-slice. This is because the scheduling order of the thread depends solely on its position in the bit-vector holding the masked requests of the current sub-round, whereas the length of the time-slice is determined by its priority only. The masked requests that have been forwarded to the internal RRA circuit cannot be updated after a sub-round has begun, while the new priority value is accessible immediately to the system timer that determines the length of a time-slice. Otherwise, the thread will be scheduled simply based on its new priority, resulting in a different scheduling order.

The main difficulty for classical PQ architectures to be shared among multiple thread queues is that the threads held in them are sorted in a FIFO order[12]. Therefore, upon a queue switch, all entries in the current queue should be exported in this order so that they could be loaded back in the same order later. None of these PQ architectures provides a quick access interface. As a result, all these threads could be only accessed sequentially, causing that the time taken by a queue switch would be varied depending on the sizes of both queues. In contrast to them, a thread queue of the WRRA-PQ architecture is just a bit-vector and so is its internal mask register. Basically, when using a WRRA-PQ, switching thread queues simply means switching both bit-vectors, which can be executed in constant time.

Figure 4.34 demonstrates the simplified architecture used in the monitor manager, which allows multiple monitors to share a single WRRA-PQ. Through the multiplexer on the left side, we can read either a thread queue of a monitor or its mask out of the monitor table and load it into an intermediate thread queue circuit by performing the enqueue operation. Each time before loading a new value into the thread queue circuit, we first need to reset it to ensure that the input value is not corrupted by the legacy data in it. This causes that the WRRA circuit is also reset at the same time.

A key thing to note is that we do not update the mask register inside the WRRA circuit directly[13]. Instead, during a scheduling process, we first use the mask of the queue as input for the WRRA, if it

[12] The BTC-PQ is excluded from this discussion.

[13] The WRRA circuit does not have a dedicated input port for updating mask value.

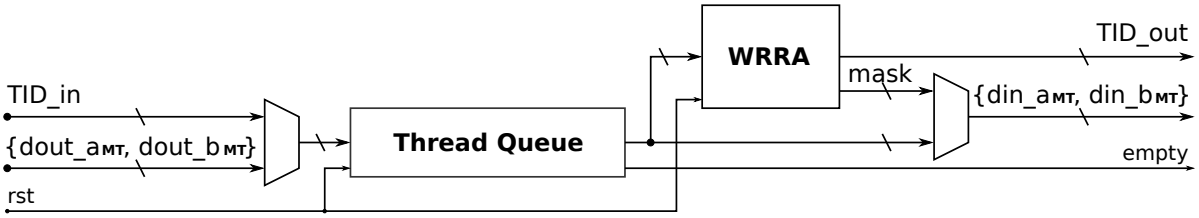


Figure 4.34: Circuit for sharing a WRRA-PQ among monitors

is not empty. This is because the mask actually holds all remaining threads of the current scheduling round. After a thread has been selected, we simply write the updated mask value of the WRRA back into the monitor table through the multiplexer on the right side. Then, we load the corresponding thread queue into the intermediate thread queue circuit and remove the thread just selected from it. After that, the updated thread queue is written back into the monitor table. However, if the mask is empty, we need to start a new scheduling round by using the thread queue as input for the WRRA directly.

Utility Components

In the following, we give a brief overview of the important utility components used in the datapath of the scheduler.

one-hot Encoder and Decoder: The one-hot encoder converts a n-bit one-hot value oh to a m-bit unsigned binary value b, where $m \leq 2^n$, as follows:

$$b = i, \text{ if } oh[i] = 1 \text{ and } i \in [0, n-1] \tag{10}$$

and the one-hot decoder executes the inverse operation, which is formally defined below:

$$oh[i] = 1, \text{ if } b = i \text{ and } i \in [0, 2^m - 1] \tag{11}$$

Both circuits are frequently used to convert thread/monitor IDs between the one-hot and binary formats.

Attribute Table: This circuit implements a general-purpose data table by adopting BRAMs. It is employed to realize both thread and monitor tables in the thread scheduler. Like a BRAM, it also has dual ports that allow for random accesses to two arbitrary attributes at the same time. An attribute needs to be addressed using two values: a thread/monitor ID and an attribute number. Except this specific addressing scheme, this circuit is primarily the same as a BRAM in the true dual-port mode, from timing to interface signals.

Content-Addressable Memory: Just as its name indicates, a CAM returns an address at which a given value is stored. It contains a database of address values, each of which is indexed by its content value. Through comparing an input content value with the existing ones in parallel, the corresponding address value can be located efficiently. The scheduler of the AMIDAR processor exploits a CAM to convert an object handle to an index of the monitor table. Through the index returned by the CAM, the fat lock data structure of the monitor can be easily accessed. Currently, we simply utilize the CAM IP core provided by Xilinx. An online reference manual describes this IP core in more detail [125].

ALU: This is a simple 10-bit ALU with two operand ports: a and b, as well as a result port r. Additionally, it has an 1-bit status output signal s. This ALU only supports a small number of arithmetic and logical operations, which are used to facilitate thread scheduling and synchronization. All these operations are described briefly in Table 4.24. This ALU allows for a high degree of customization such as removing unused operations or changing the data width of individual operations. Currently, the 10-bit data width is determined by the 3 bitwise operations which are adopted to realize the priority inheritance protocol. To simplify the manipulation of thread priorities, the one-hot format is exploited. Since 10 priority levels are defined in Java, a priority value in one-hot form needs a total of 10 bits to be represented. Also, an important thing to note about INCR and DECR is that both operations are solely used for updating the recursive lock count of a monitor. Upon the assumption that locking a single monitor in a nesting depth larger than 1024 is barely possible, the data width of these two operations is also limited to 10 bits. However, as mentioned above, each operation of the ALU can be easily adapted to various requirements as desired.

Operation	Description
INCR/DECR	increments/decrements a by 1
OR/AND/XOR	performs a bitwise OR/AND/XOR operation on a and b
EQ	asserts s if a is equal b
GT	asserts s if a is greater than b

Table 4.24: Operations of the ALU in the thread scheduler

System Timer: This circuit consists of three major parts: a system time counter, a time-slice counter and a comparator. The system time counter increments its value by 1 each clock cycle and allows the value to be read out at the software level through a system API method. This method converts the system time in cycles into the real world time according to the clock frequency. The time-slice counter is initialized upon a context switch and decrements its value each clock cycle. Once the value becomes 0, a *system tick* is generated and sent to the scheduler controller which switches the context of the processor from the current thread to the next one. The initial value of the counter (i.e. the length of the new time-slice) is determined by the priority of the next thread, using a quite simple scheme: the time-slice of the priority $p+1$ is twice as long as that of the priority p, where $p \in [1,9]$. This scheme can be easily realized with a shift operation in hardware. The length of the time-slice of the lowest priority (i.e. priority 1) is customizable via a parameter. The comparator compares a given time-out value with the current system time and asserts a status flag if the time-out value has expired.

4.6.2 Hardware-Software Interface

As mentioned in Section 2.2.2, Java provides a number of native methods to support multi-threading at the language level directly. These methods serve as the interface between the thread scheduler and a Java program. Thus, they must be able to be executed across the hardware-software boundary. In the following, we first explain the functional unit native interface (FU-NI) mentioned in Section 4.1.4 in general and then discuss how to exploit it to implement the thread- and synchronization-specific native methods in particular. At the end of this section, we explain how a thread is created and terminated at both software and hardware levels.

Functional Unit Native Interface

The native interface of an FU is implemented at three different levels. At the hardware level, the FU should provide a set of operations that are required when running Java programs. At the software level, a static native method should be declared through which the operations built into the FU can be invoked. The parameter and return types of the method are determined by the operations supported by the FU. In certain circumstances, explicit type conversions may be necessary. Note that the type of the first parameter of this method is restricted on **`int`**. To provide a descriptive, consistent naming convention, the name of the method should begin with **`invoke`** and end with the name of the FU, e.g. **`invokeScheduler`**. The bytecode assigned to the native method serves as the intermediate level between the FU and the native method. Upon generating the AXT file for a Java program, it is patched to replace all invocations of the native method that occur in the bytecode streams of the program. At runtime, the token set of the bytecode is executed consequently at the positions where the method is called. The token set contains primarily just two parts:

- Sending all operands from the frame stack to the target FU.
- Sending an opcode to the FU, which triggers the target operation.

A key thing to note is that the opcode which is sent to the target FU does not represent any operation implemented in the FU. It is actually a meta-opcode called `INVOKE`. Every FU that implements the native interface needs to support this opcode. An operation that needs to be executed by the FU indeed is encoded using the first integer parameter of the native method.

Native Interface of the Thread Scheduler

To better illustrate the FU-NI, we describe how the native interface of the thread scheduler is implemented in a top-down fashion. At the Java level, all native methods are declared in class **`de.amidar.AmidarSystem`** for ease of management. Principally, any FU that implements the FU-NI should be associated with a single native method in order to limit the number of patched bytecodes. However, two native methods are declared for the thread scheduler to realize all thread- and synchronization-specific methods described in Section 2.2.2 properly. This is because invoking some of these methods suspends the execution of the current thread and causes a context switch immediately. These methods are referred to as blocking methods in the following. Examples include the **`sleep`**- and **`wait`**-methods. As a blocking method is invoked, not just the thread scheduler but also the token machine needs to be signaled so that token machine can halt its internal pipeline to avoid loading the bytecode following the invocation of the method further. Listing 12 shows both native methods of the scheduler.

Listing 12: Native methods of the thread scheduler

```
// blocking invocation
public static native void invokeBlkScheduler(int opcode,   int operand0,
                                              int operand1, int operand2);
// nonblocking invocation
public static native int invokeScheduler(int opcode,   int operand0,
                                         int operand1, int operand2);
```

The former native method is employed for the implementation of blocking methods, while the latter for the implementation of nonblocking methods. This indicates that invoking the former method results in a context switch.

Upon generating an AXT file, the invocations of both native methods are replaced with two unused bytecodes that are referred to as **invokeblkscheduler** and **invokescheduler** respectively. Listing 13 and 14 illustrate their token sets. Note that **invokeblkscheduler** is declared as an unconditional jump instruction by using `#JUMP_BYTECODE`. This causes that the token machine stops fetching new bytecode from the position addressed by the current PC value. The first two tokens of both bytecodes are identical and just used to transport the 4 integer parameters of the corresponding native methods from the frame stack to the scheduler. Then, both **invokeblkscheduler** and **invokescheduler** deliver the meta-opcode `INVOKE` to the scheduler in their third tokens. Consequently, the decoder of the scheduler fetches the actual opcode from the data register holding the lower 32 bits of the operand received by the first input port. Note that the last two tokens of **invokescheduler** are interdependent. This is because that opcode `PUSH32` contained in the fourth token can only be carried out by the frame stack after the scheduler has completed its operation and sent its result to the frame stack, i.e. `INVOKE` and `PUSH32` must be executed sequentially. In contrast, the operations associated with the last two tokens of **invokeblkscheduler** can be executed in parallel, since `FORCESCHEDULING` does not have any operand. Once the token machine receives `FORCESCHEDULING`, it asserts a signal called $CS\_waiting$, which causes the scheduler to start a context switch process. During the context switch, the token machine obtains the PC and AMTI of the next thread from the scheduler and starts fetching bytecodes from the new position determined by both values. The context switch process is introduced in Section 4.6.3 in more detail.

Listing 13: Token set of invokeblkcheduler

```
0: invokeblkscheduler
1: #JUMP_BYTECODE
2: {
3:   T(framestack,   POP64, scheduler.1),
4:   T(framestack,   POP64, scheduler.0),
5:   T(scheduler,    INVOKE);
6:   T(tokenmachine, FORCESCHEDULING)
7: }
```

Listing 14: Token set of invokescheduler

```
0: invokescheduler
1: {
2:   T(framestack, POP64,  scheduler.1),
3:   T(framestack, POP64,  scheduler.0),
4:   T(scheduler,  INVOKE, framestack.0)++;
5:   T(framestack, PUSH32)
6: }
```

Like every FU, the scheduler also has a standard AMIDAR infrastructure interface (AII), through which tokens and data can be delivered to it. Once a token arrives, it needs to be preprocessed by the operation decoder that extracts the opcode held in it. Besides the meta-opcode `INVOKE`, there are also two regular opcodes supported by the scheduler, namely `MONITORENTER` and `MONITOREXIT`, which correspond to bytecodes **`monitorenter`** and **`monitorexit`** respectively. If a token contains either of these opcodes, the opcode is forwarded to the operation executor directly. Otherwise, the decoder reads out the actual opcode from the corresponding data register of the first input port. Another important task of the decoder is to receive the operands of an operation and redirect them to the scheduler datapath. Both `MONITORENTER` and `MONITOREXIT` have a single operand, namely the handle of a monitor object. Although the meta-opcode INVOKE is delivered with 4 operands together, the actual opcode does not always require all of them. Thus, for these opcodes, the decoder forwards only the actually needed operands to the scheduler datapath. Note that `MONITORENTER` and `MONITOREXIT` can also be invoked through the native interface of the scheduler, which is quite useful for realizing the **`wait`**-method.

Table 4.25 and 4.26 list all operations supported by the scheduler, which are employed to re-implement the important thread- and synchronization-specific methods. In Table 4.27, the remaining operations of the scheduler are illustrated, which enable interacting with the scheduler at the software level directly. Note that all operand and result types shown in the tables correspond to the original types declared in Java. However, as Listing 12 demonstrates, the types of all parameters of the two native methods are restricted on **`int`**. This means that reference types like **`Thread`** and **`Object`** must be converted to integers via the native **`refToInt`**-method mentioned in Section 4.1.4.

Opcode	Operands	Result
`NEW`	`Thread t,int amti`	int tid
`SET_PRIORITY`	`int tid,int prio`	void
`SET_DAEMON`	`int tid`	void
`START`	`int tid`	void
`SLEEP`	`int time_l,int time_h`	void
`YIELD`	`int tid`	void
`INTERRUPT`	`int tid`	void
`TERMINATE`	`int tid`	void

Table 4.25: Operations supporting thread-specific methods

Opcode	Operands	Result
`MONITORENTER`	`Object m`	void
`MONITOREXIT`	`Object m`	void
`WAIT`	`Object m`	void
`TIMED_WAIT`	`Object m,int time_l,int time_h`	void
`NOTIFY`	`Object m`	void
`NOTIFYALL`	`Object m`	void

Table 4.26: Operations supporting synchronization-specific methods

Opcode	Operands	Result
ENABLE_CS	none	void
DISABLE_CS	none	void
CURRENT_THREAD_ID	none	int tid
SYSCLK_LOW	none	int sys_time_l
SYSCLK_HIGH	none	int sys_time_h
READ_MONITORTABLE	Object m,int attrid	void
READ_THREADTABLE	int tid,int attrid	void

Table 4.27: Operations supporting interacting with the thread scheduler

The majority of the operations shown above are self-explaining. Thus, we only describe ENABLE_CS and DISABLE_CS contained in Table 4.27 briefly. The former operation activates context switch, while the latter deactivates it. A key point is that only the context switch controller shown in Figure 4.23 is affected by both operations, not the entire scheduler. After the execution of DISABLE_CS, a new thread is still allowed to be created and even be started. However, it cannot really start running on the processor until ENABLE_CS is executed. This is especially relevant for creating ISTs, which is performed by the bootloader of the AMIDAR processor, because context switch is deactivated by default after system reboot to ensure the whole system to be initialized properly. Also, these two operations provide an optional synchronization mechanism in addition to the classical monitor-based mechanism. A critical section that needs to be performed atomically by a single thread can be enclosed by DISABLE_CS and ENABLE_CS, ensuring that no other thread can preempt the current one before it leaves the critical section.

Creation and Termination of a Thread

Figure 4.35 shows the entire lifetime of a thread `t` from its creation to its termination. Throughout the whole process, three classes are used frequently, namely **java.lang.Thread**, **de.amidar.Scheduler** and **de.amidar.AmidarSystem**. Class **Scheduler** is the abstract representation of the scheduler at the software level, which provides a user-friendly programing interface that hides the low-level details of the native interface of the scheduler. Only a single instance of this class can be created, ensuring the $1:1$ mapping relationship between hardware and software.

As Figure 4.35 illustrates, a thread instance can be simply created through the invocation of the constructor of class **Thread**, just like in any other Java runtime systems. At the beginning of the constructor, the **creatThread**-method is called on the singleton instance of class **Scheduler** as follows:

Listing 15: Creating and initializing a thread in the AMIDAR processor

```
tid = Scheduler.Instance().createThread(this, NORM_PRIORITY);
```

The ID returned by this method corresponds to the index of the newly created thread into the thread table of the scheduler. Almost all thread-specific operations supported by the scheduler need to be invoked through this ID. Besides initializing a new thread in the scheduler, the **createThread**-method also initializes a dedicated Java stack for the thread, which is described later below.

Since the two native methods of the scheduler may only accept integer parameters, the first parameter of the **createThread**-method, i.e. the thread instance being created, must be first cast to an integer,

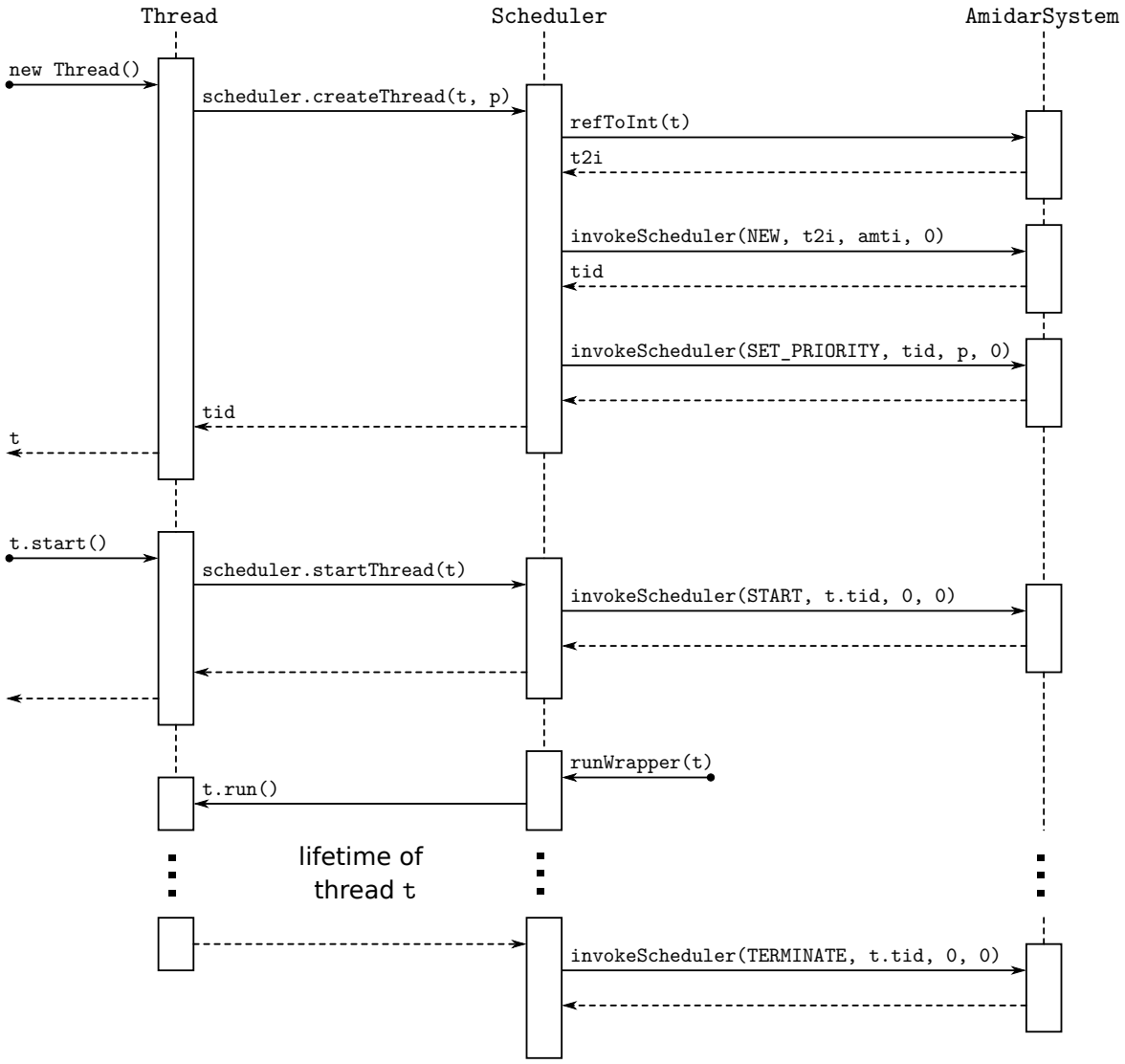


Figure 4.35: Creation and termination of a thread

using the **refToInt**-method. After that, the NEW operation is performed on the scheduler through calling the **invokeScheduler**-method. If there is still an unused entry in the thread table, this operation yields the index of this entry, which is then used as the ID of the newly created thread; otherwise, it returns -1 back. In the latter case, the current thread that tries to create the new thread enters a spin-waiting loop and waits until a free entry is available.

Once the new thread obtains a valid ID, it is assigned a default priority[14]. Note that the priority of a thread is held in both priority table and thread table of the scheduler. In the priority table, it stays in binary form to reduce the usage of on-chip registers and increase the performance of the WRRA circuit. In contrast, the thread table stores it in one-hot form for ease of implementation of the priority inheritance protocol. This implies that the priority of t needs to be converted to the one-hot format inside the scheduler before it is entered into the thread table.

After the new thread has been assigned a priority, it is ready to be scheduled from the viewpoint of the scheduler. However, to be really capable of running on the processor, the thread also needs a

[14] The default priority is 5 currently.

dedicated Java stack. Through the WB interface of the frame stack, the Java stack is initialized as shown in Listing 16.

Listing 16: Initializing the Java stack

```
1: // AmidarSystem.invokeScheduler(DISABLE_CS,0,0,0);
2: synchronized(frameStack){
3:   // initialize the Java stack of t
4:   frameStack.threadSelect         = tid;
5:   frameStack.localsPointer        = 0;
6:   frameStack.callercontextPointer = 1;
7:   frameStack.stackPointer         = 1+FrameStack.CALLERCONTEXT_WORDS;
8:   frameStack.maxPointer           = frameStack.stackPointer;
9:   frameStack.overflow             = 0;
10:  // manually write t to address 0 of the Java stack
11:  frameStack.stackAddressSelect   = 0;
12:  frameStack.stackData            = t2i;
13:  frameStack.stackMeta            = FrameStack.ENTRYTYPE_REF;
14: }
15: // AmidarSystem.invokeScheduler(ENABLE_CS,0,0,0);
```

The key goal of the code above is to manipulate the first frame of the Java stack in such a way that a method seems to be already invoked by `t` even before `t` is started. This method has a single parameter, namely the thread `t` itself, and does not have any return value. To meet this goal, we first locate the pointers associated with `t` that are held in the context table via the ID of `t` at line 4. Then, a caller context for the method is created by manually assigning each of these pointers a specific value from line 5 to line 9. First, the **`localsPointer`** is set to 0, indicating the location of the first and only local variable of the method, which corresponds to the method's single parameter. Since there is only one local variable, the caller context starts at address 1, which is set at line 6. Accordingly, the current **`stackPointer`** is set to the first location after the caller context at line 7. At line 8, **`maxPointer`** that is used to track the maximum stack depth is set to the same location pointed by **`stackPointer`**, and the exception flag **`overflow`** is reset, indicating that no overflow has occurred on this Java stack so far. After the caller context has been configured, we set the offset register **`stackAddressSelect`** to 0 so that thread `t` and its data type can be written to address 0 of the Java stack at line 12 and line 13 respectively. Since multiple threads can access the **`frameStack`** object simultaneously, the code described above is enclosed in a synchronized block entirely, to avoid race conditions. Note that we could also exploit the `DISABLE_CS` and `ENABLE_CS` operations provided by the scheduler to protect this critical section.

The reason why a method needs to be initialized in the Java stack is that every thread requires an entry point from which it can start the execution of its task. Since the entry point method is the first-ever method executed by a thread, it does not have any caller. As a result, its local variable memory and caller context cannot be automatically established by the frame stack as usual, and therefore must be created manually, as described above. However, this is still not enough for the invocation of the entry point method. Additionally, we must provide its position in the method table so that the token machine

can fetch its bytecodes properly. Thus, upon executing the `NEW` operation on the scheduler, the AMTI of this method is delivered to the scheduler as the second operand, as shown in Figure 4.35.

According to the Java specification, the standard entry point of a thread should be its **run**-method. However, a thread instance can be created from an arbitrary subclass of **Thread**, causing that the AMTI of the **run**-method can be varied from one thread to another. To solve this problem, we declare a static wrapper method with a single parameter in class **Scheduler** and use it as the common entry point for all threads. Listing 17 demonstrates this method in detail.

Listing 17: Thread entry point method

```
 1: static void runWrapper(Thread t) {
 2:   try {
 3:     t.run();
 4:   }
 5:   catch (Exception e) {
 6:     if (System.err != null) {
 7:       System.err.println("Uncaught exception in Thread "+
 8:                          t.getName()+":");
 9:       System.err.println(e.getMessage());
10:       e.printStackTrace(System.err);
11:     }
12:   }
13:   // clean up the context of the thread being terminated
14:   terminateThread (t);
15: }
```

The method above invokes the **run**-method on its single parameter of type **Thread**, which hides the differences among various subclasses derived from class **Thread**. Note that this parameter corresponds just to the thread that we manually write to the address 0 of a new Java stack. One of the major advantages of this solution is that all threads have one common entry point method whose AMTI can be easily determined, since it is a static method. Another advantage is that we can handle exceptions thrown from the **run**-method explicitly, increasing the program robustness. In addition, we can clean up the context of a terminated thread thoroughly, avoiding its legacy data to interfere with the further program execution.

After a thread has been created and initialized at both hardware and software levels, it is still not taken into account by the scheduler until its **start**-method is invoked. At some later time point after that, the scheduler will select this thread to run on the processor and provide its ID, AMTI as well as PC to the token machine. Note that the PC of a new thread is always set to 0 to ensure the **runWrapper**-method to be executed from the beginning.

Internal Representation of a Thread

Inside the scheduler, a thread can be considered as a data structure consisting of 14 different attributes. These attributes are described in Table 4.28 briefly, where $n_m = log_2 N_m$ and N_m represents the maximum number of monitors allowed to be used simultaneously.

No.	Attribute	Width	Description
0	HANDLE	32-bit	The handle of the thread object.
1	AMTI	16-bit	The AMTI of the method being executed by the thread.
2	PC	16-bit	The PC of the method being executed by the thread.
3	PRIORITY	10-bit	The priority of the thread in one-hot form.
4	STATE	8-bit	The state of the thread in one-hot form.
5	TIMEOUT_L	32-bit	The low 32-bit timeout value.
6	TIMEOUT_H	32-bit	The high 32-bit timeout value.
7	NEEDED_MONITOR	n_m-bit	The monitor the thread requires but does not acquire.
8	LOCK_CNT	10-bit	The lock count of the monitor the thread is waiting for.
9	MONITOR_PRIORITY	10-bit	The one-hot priority of the monitor the thread is waiting for.
10	MONITOR_PREVIOUS	n_m-bit	The previous monitor of the monitor the thread is waiting for.
11	MONITOR_NEXT	n_m-bit	The next monitor of the monitor the thread is waiting for.
12	MONITOR_CHAIN_HEAD	n_m-bit	The first monitor the thread acquires.
13	MONITOR_CHAIN_TAIL	n_m-bit	The last monitor the thread acquires.

Table 4.28: Thread attributes

These attributes can be further partitioned into two groups: basic and synchronization-specific attributes. The former group includes the first 7 attributes, while the remaining 7 ones belong the latter group. The discussion in this section below refers to the basic attributes only. In the following section, the other attributes are described in more detail.

All attributes of a thread are solely held in the thread table except its priority that is also saved in the priority table. Each of them can be accessed separately by combing the thread ID with the attribute number. Since the thread table has dual ports, we can access two attributes of two different threads at a time.

The priority table holds the current priorities of all threads in binary form and outputs them in parallel to the two WRRAs built into the scheduler datapath as shown in Figure 4.23. If the priority of a thread is changed due to either the invocation of the **`setPriority`**-method or the priority inheritance at runtime, the corresponding entries in both tables must be updated accordingly. In the former case, the priority values stored in both tables should be equal after the update, if they were converted to the same format. In the latter case, the one-hot value in the thread table contains both original and inherited priorities at the same time, whereas the binary value in the priority table is altered to the inherited priority. For example, given a newly created thread with the default priority, the priority value saved in the thread table is $00\_0001\_0000_2$ and the corresponding value saved in the priority table is 5_{10}. Assume that the thread inherits the priority 8 sometime, the priority value in the thread table is updated to $00\_1001\_0000_2$ and that in the priority table is changed to 8_{10}. A key thing to note is that invoking

the **setPriority**-method does not alter the priority of a thread, if the thread has inherited at least a priority. In Section 4.6.4 below, we explain the implementation of the priority inheritance in detail.

Thread Queues

The scheduler utilizes a total of 6 instances of the thread queue circuit described in Section 4.6.1 to manage threads in various states. In the following, we first give detailed information about each of them. Then, we introduce the *speculative dequeue* operation that is used frequently to simplify the management of multiple copies of a single thread in different thread queues. Note that the blocked and waiting thread queues of a monitor are stored in the monitor table directly rather than in dedicated thread queue instances.

- ***Free Thread Queue:*** This queue contains all free thread slots in the scheduler. In this context, a thread slot means an unused entry in the thread table[15]. After system reboot, all of the bits in the queue are set to 1 except the least significant bit (i.e. the 0th bit). The 0th thread slot is reserved for the main thread that starts running immediately after system reboot. To select a free slot for each newly created thread, an arbiter is exploited, which always yields the currently leftmost nonzero bit in the queue. For example, given a customized AMIDAR processor with support for up to 8 threads, its free thread queue should be 1111_1110_2 after system reboot. Then, we create two thread instances, namely T_1 and T_2 in sequence. Consequently, T_1 obtains ID 7_{10}, while T_2 ID 6_{10}. After removing their IDs in one-hot form from the free thread queue (i.e. both threads are dequeued), the queue becomes 0011_1110_2 and still contains 5 free slots.

- ***Ready Thread Queue:*** This queue contains all runnable threads. A thread is said to be runnable, if it can start running on the processor as soon as it is assigned a time-slice, without requiring any other extra resource like a monitor. Note that a newly created thread is not held in this queue until the **start**-method is invoked on it. As discussed in Section 4.6.1, regular user threads that perform periodic tasks can be scheduled in their starting order only if their starting order coincides their spatial order in the ready thread queue. According to the description about the free thread queue above, the spatial order of a thread in the ready thread queue is determined by the order in which it is created. Therefore, if all user threads are started exactly in their creation order, they will always be scheduled in their starting order[16].

- ***Daemon Thread Queue:*** A thread is added into this queue, if the **setDaemon**-method is invoked on it. An application terminates after all non-daemon threads have terminated.

- ***Timed-waiting Thread Queue:*** A thread is added into this queue, if the **sleep**-method or the **wait**/**join**-method with a timeout value is invoked on the thread.

- ***Waiting IST Queue:*** All waiting ISTs are held in this queue. The AMIDAR processor uses a novel *wait-interrupt*-based interrupt handling model. The basic idea of this model is that an incoming

[15] Note that the priority table can be considered as a logical part of the thread table because each of its values is always synchronized with the corresponding one in the thread table passively and there is no other way to change any of its values independently.

[16] This discussion is focused on the most typical application scenario of embedded systems, without concern for more general and complicated use cases such as thread synchronization.

interrupt request can be handled only if its IST is waiting for its occurrence. This means that the interrupt handler needs first to check the waiting IST queue to ensure whether the corresponding IST is ready, as an interrupt request arrives. If the IST is still handling the previous interrupt request (i.e. it is not in the waiting IST queue), the interrupt handler ignores the incoming one until the IST completes its current interrupt service routine. We give more details about the interrupt handling mechanism used by the AMIDAR processor in Section 4.6.5.

- ***Started Thread Queue:*** This queue contains all threads alive. A thread is alive, if it has been started and not been terminated yet [49]. One of the usages of this queue is to expose information about currently alive threads to the debugger. Also, we compare this queue with the daemon thread queue continuously. Once both queues become equal, i.e. all threads alive are daemon threads, an internal status signal is asserted, which halts the execution of the whole processor.

An important thing to note about the thread queues described above is that a single thread can be held in multiple thread queues. For example, a daemon thread exists in the ready, started and daemon thread queues at the same time. Once the thread terminates by invoking the **`terminateThread`**-method on itself, we can remove it from both former queues directly. This is because a thread running on the processor must be both runnable and started. However, without an explicit check, we are not sure about whether the thread is also in the daemon thread queue, since being a daemon thread is an optional property. To simplify the control logic, we avoid the extra check in such a way that we first add the thread into and then remove it again from the daemon thread queue, using the bitwise OR- and XOR-operations respectively. If the thread was already in the queue, it is removed from the queue anyway; if not, the queue remains unchanged. In the following, we refer to the combination of an enqueue and dequeue operation that are performed successively on a single thread as *speculative dequeue*. This operation is especially useful for executing the **`notifyAll`**-method because we do not have to distinguish between waiting and timed-waiting threads and only need to speculatively dequeue all of them from the timed-waiting thread queue once. This avoids the need for separately checking each of them completely.

Thread States

One of the key tasks of the scheduler is to maintain states of threads. It needs to perform a state transition for a thread as a specific event occurs. In Java, such an event is primarily caused by executing one of the methods described in Section 2.2.2. Figure 4.36 demonstrates the thread states used in the scheduler and the major transitions among them. An important thing to note is that the **`join`**-method is not shown in the figure, because it has been implemented solely based on the **`wait`**-method[17]. Upon a state transition, the scheduler also needs to perform corresponding enqueue/dequeue operations to ensure that every thread is held in correct thread queue(s).

The Java specification defines 6 standard thread states: *NEW*, *RUNNABLE*, *BLOCKED*, *WAITING*, *TIMED_WAITING* and *TERMINATED*. Inside the scheduler, we add two extra states: *SLEEPING* and *IOING*, since we need additional state information for ease of implementation of some operations. An 8-bit binary value is used to represent the states of a thread in which each bit corresponds to a specific state. The state bit for *SLEEPING* can only be set if the bit for *TIMED_WAITING* is set. Similarly, the state bit

[17] Almost all mainstream JDKs use a similar implementation, including Oracle JDK and OpenJDK.

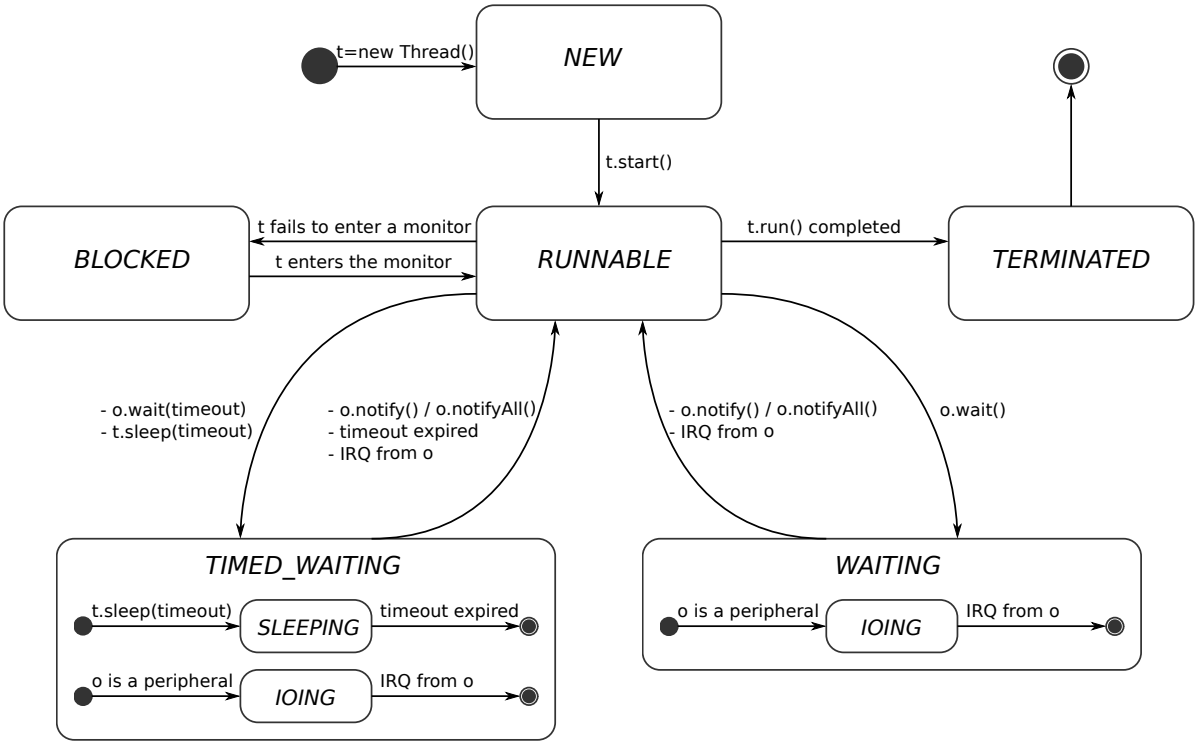


Figure 4.36: Thread state transitions

for *IOING* can be set only if the thread is either in the state *WAITING* or in the state *TIMED_WAITING*. In the following, we describe each of these states in more detail.

- ***NEW:*** This is the state for a thread that has been created but not started. A thread may be in this state only once in its entire lifetime. No thread queue contains a thread in this state.

- ***RUNNABLE:*** After a new thread is started, its state becomes *RUNNABLE*. Only a runnable thread is allowed to run on the processor. A thread stays in this state until it terminates or is suspended for some reason. It may leave this state for multiple times in its lifetime, but must be able to return to the state if it is still alive. A runnable thread is held in both ready and started thread queues.

- ***BLOCKED:*** This state is for a thread that tries to acquire a monitor but fails. Once a thread obtains the needed monitor, its state is transitioned back to *RUNNABLE* immediately. A thread in this state is saved in the blocked thread queue of the needed monitor as well as in the started thread queue.

- ***WAITING:*** This state indicates that a thread is waiting on an object for its monitor without time limit. A waiting thread can be awakened by the invocation of the **`notify`**- or **`notifyAll`**-method on the object. Although an awakened thread becomes runnable, it cannot proceed and return from the **`wait`**-method until it reenters the needed monitor. A waiting thread is contained in the waiting thread queue of the needed monitor and the started thread queue at the same time. If the object on which the **`wait`**-method is invoked corresponds to a peripheral object[18], the extra state bit *IOING* also needs to be set. In this case, the waiting thread is an IST and must be inserted into the waiting IST queue additionally. A waiting IST is awakened by an interrupt request from the

[18] A peripheral object is the abstract representation of a hardware peripheral device at the software level.

peripheral device. Technically, it could also be awakened through the invocation of the **notify**- or **notifyAll**-method on the peripheral object. However, this does not fit into the interrupt handling model used by the AMIDAR processor and therefore should be avoided.

- ***TIMED_WAITING:*** A thread enters this state by invoking either the **wait**-method with a timeout value or the **sleep**-method. In the former case, this state is basically equivalent to *WAITING*, except that a timed-waiting thread is held in the timed-waiting thread queue in addition and wakes up by itself upon the expiration of the timeout value. In the latter case, the extra state bit for *SLEEPING* needs to be set, to distinguish a sleeping thread from a thread timed-waiting on an object for its monitor. This is because a sleeping thread only needs to be removed from the timed-waiting thread queue and added back into the ready thread queue after its sleep time has elapsed. In contrast, a timed-waiting thread needs to be removed from the waiting thread queue of its needed monitor additionally. A sleeping thread is held in both timed-waiting and started thread queues.
- ***TERMINATED:*** After a thread has completed execution of its **run**-method, it is moved to this state. A terminated thread is added back to the free thread queue, allowing its thread slot to be reused.

Thread Scheduling

One of the major tasks of the system routine executor shown in Figure 4.23 is to select the next thread by exploiting the WRRA connected to the thread manager. The selecting process is controlled by a simple FSM. After a context switch or an enqueue operation has just been performed on the ready thread queue, this process is triggered under one single condition: the next TID buffer is still empty. This buffer is implemented using a simplified stack circuit of depth 2, i.e. it can hold up to two thread IDs once. However, the thread selecting process can only push one into the buffer because it cannot be triggered anymore if the buffer is not empty. The other buffer entry is reserved for the operation executor.

The WRRA only has the ready thread queue as input. Its output first needs to be converted to a binary value and then pushed into the next TID buffer. Note that the selected thread stays in the ready thread queue further because a thread can be removed only upon a state transition as described above. There are two special cases in which the selected thread is considered invalid and needs to be abandoned:

1. The ready thread queue contains only one thread. As a result, the next thread selected by the WRRA is equal to the current one. To avoid an unnecessary context switch, the ID of this thread is not written into the next TID buffer.
2. The ready thread queue does not contain any thread, which can happen, e.g. when all threads are sleeping. This case can be easily identified by checking the $empty$ signal of the ready thread queue.

The next TID buffer is cleared solely during a context switch. Thus, most of the time, the thread selecting process is performed immediately after a context switch. However, if no valid thread can be found, the next TID buffer remains empty. As a result, once a thread becomes runnable (i.e. is enqueued into the ready thread queue), the thread selecting process is triggered and then pushes this thread into

the next TID buffer directly without the need for a validity check. Note that this newly enqueued thread can be equal to the current one in the second case above.

Management of Timed-waiting Thread

Another major task of the system routine executor is to wake up timed-waiting threads. Like the thread selecting process above, the thread awakening process is also controlled by an FSM. It is started if the timed-waiting thread queue is not empty and the scheduler happens to be idle, i.e. no other task is being performed, such as selecting thread or switching context etc.

In this process, the timeout value of each thread held in the timed-waiting thread queue is checked in a round-robin manner, by exploiting a RRA. Note that a relative timeout value in milliseconds which is given at the Java level first needs to be transformed to an absolute timeout value in clock cycles based on the current system time and system clock frequency. After that, the converted timeout value is delivered to the scheduler and written into the thread table. According to the thread ID returned by the RRA, the low and high parts of the timeout value (i.e. `TIMEOUT_L` and `TIMEOUT_H`) are read out from the thread table via its dual access ports and then sent to the system timer. As described in Section 4.6.1, the system timer can determine whether a given timeout value has expired, using a built-in comparator. If the timeout value has not expired yet, the current check step is completed and the awakening process moves on to the next thread in the timed-waiting thread queue. Otherwise, the current thread is awakened as follows, where `tid`, `rtq` and `twtq` represent the ID of the thread that needs to be awakened, the ready thread queue and timed-waiting thread queue respectively.

Algorithm 5: Awakening a thread

input : tid, rtq, twtq
output: void

```
1  state = readThreadTable(tid, STATE)
2
3  // thread is timed-waiting on an object
4  if state & "SLEEPING" == 0 then
5      mid = readThreadTable(tid, NEEDED_MONITOR)
6
7      // dequeue t from the waiting thread queue of the needed monitor
8      tq = readWaitingThreadQueue(mid)
9      dequeue(tq, tid)
10     writeWaitingThreadQueue(mid, tq)
11
12     // dequeue t from the waiting thread queue mask of the needed monitor
13     tq = readWaitingThreadQueueMask(mid)
14     speculativeDequeue(tq, tid)
15     writeWaitingThreadQueueMask(mid, tq)
16 end
17
18 dequeue(twtq, tid)
19 enqueue(rtq, tid)
20 writeThreadTable(tid,STATE,"RUNNABLE")
```

To awaken a thread, we need to remove it from the timed-waiting thread queue (line 18) and then add it back to the ready thread queue (line 19). Also, the state of the thread should be changed to *RUNNABLE* (line 20). Additionally, for a thread timed-waiting on an object for its monitor, it must be removed from the waiting thread queue of the monitor, by exploiting the thread queue circuit shown in Figure 4.34. Note that we are not sure about whether the thread is contained in the mask of the waiting thread queue, because this depends on when the current scheduling round begins and when the thread starts waiting on the object. To avoid an extra check, we delete it from the mask speculatively (line 14). Awakening a thread must be performed atomically, i.e. no other task may interrupt it.

Context Switch

A context switch can be caused by three different events, namely the occurrence of an interrupt, the assertion of the *CS_waiting* signal as well as the expiration of the current time-slice (i.e. system tick). The first event causes that an IST is started, where the ID of the IST is determined by the interrupt selector shown in Figure 4.23. The second event is triggered when the `FORCESCHEDULING` operation is performed on the token machine. This provides programmers the capability to explicitly request a context switch at the software level. The last event comes from inside the scheduler and is the basis for implementing the preemptive thread model of Java. If the next TID buffer in the thread manager is empty, the last event is ignored to avoid unnecessary context switches.

If one of the three events occurs, the context switcher asserts the *CS_request* signal and an internal busy flag at the same time. Upon assertion of *CS_request* or execution of `FORCESCHEDULING`, the token machine stops fetching new bytecode immediately and starts checking the statuses of the token adapters of all FUs continuously. After all token adapters become empty, the token machine outputs the data of the current thread (i.e. AMTI and PC) and sets the *CS_ready* signal to acknowledge the request from the scheduler. In the meantime, due to the assertion of the internal busy flag, the operation executor and system routine executor of the scheduler attempt to halt executing their current tasks at a safe point as soon as possible. For the operation executor, a safe point is reached always after the opcode contained in the last token has been executed. This guarantees the atomicity of every hardware-based operation supported by the scheduler. For the system routine executor, the meaning of a safe point depends on the task being performed. If it is selecting the next thread, a safe point is reached after the selecting process has been finished. If it is awakening timed-waiting threads, a safe point is reached after the awakening process has been performed on a single thread completely.

After both operation executor and system routine executor become idle and *CS_ready* becomes active, the switch controller starts exchanging thread data with the token machine. Figure 4.37 demonstrates the handshaking protocol used for this purpose.

The switch controller first writes the AMTI and PC of the current thread into the thread table. Then, it loads both values of the next thread together with the thread's ID into the output registers of the interface to the token machine (i.e. the TMI). Simultaneously, the *CS_request* signal is cleared and a *valid* signal is set to indicate that these values are available. If a context switch is triggered by an interrupt, the ID of the IST is provided by the interrupt selector. Otherwise, the ID of the next thread corresponds to the top value of the next TID buffer. On the other side, the token machine writes the data of the next thread into its internal registers one clock cycle later and resets the *CS_ready* signal. This has the

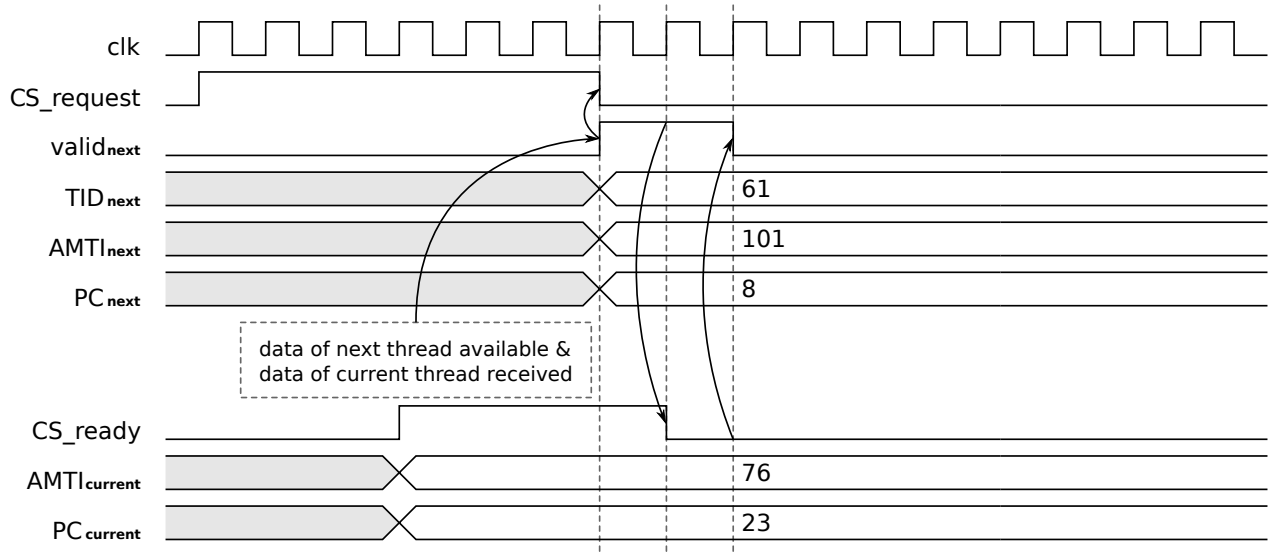


Figure 4.37: Handshaking protocol between the scheduler and token machine

consequence that the *valid* signal of the scheduler is cleared after another clock cycle. Then, the priority of the next thread is sent to the system timer which assigns the thread a time-slice based on the scheme described in Section 4.6.1. At last, the context switcher resets its busy flag, removes the next thread ID from the next TID buffer and requests the system routine executor to select a new thread if the next TID buffer becomes empty.

As mentioned above, a system tick is ignored if the next TID buffer is empty. However, if a context switch is caused by the second event, we must handle the issue of the empty next TID buffer explicitly. Such a situation can occur, e.g. when the last runnable thread of a program calls the **sleep**-method that invokes the SLEEP operation on the scheduler. In this situation, the context switcher suspends the context switch process by not setting the *valid* signal after it has received the AMTI and PC of the current thread. In addition, it clears its busy flag to allow the system routine executor to run and then starts waiting until a valid thread ID is either provided by the interrupt selector directly or pushed by the system routine executor into the next TID buffer. After that, the context switcher asserts the busy flag again. Once the system routine executor stops running, the *valid* signal is asserted, which resumes the interrupted context switch process. Note that the ID of the next thread can be equal to that of the current one in this situation.

Except for the situation described above, there is another case in which the current and next threads can be the same, namely when a thread enters a monitor successfully. In the following, we explain the reason for that in detail. The bytecode **monitorenter** is currently mapped to a single operation of the scheduler as illustrated in Listing 18.

The token set of **monitorenter** is quite similar to that of **invokeblkscheduler** shown in Listing 13 and also causes a context switch explicitly. This is because the result of executing MONITORENTER on the scheduler is nondeterministic. If the monitor is already owned by another thread, the current thread may not run further, i.e. the bytecode following **monitorenter** should not be fetched. In contrast, if the monitor is free, the current thread should be able to proceed if its time-slice has not expired yet. The former goal is achieved by invoking FORCESCHEDULING on the token machine. To meet the latter one, the

Listing 18: Token set of monitorenter

```
0: monitorenter
1: #JUMP_BYTECODE
2: {
3:   T(framestack,   POP32, scheduler.0),
4:   T(scheduler,    MONITORENTER);
5:   T(tokenmachine, FORCESCHEDULING)
6: }
```

operation executor pushes the current thread ID into the next TID buffer to avoid the previously selected thread to be started, if the current thread can acquire the monitor.

If the next thread is the same as the current one, it is unnecessary to update the time-slice counter in the system timer. Therefore, the context switcher performs an extra check on the top value of the next TID buffer before the priority of the next thread is sent to the system timer. Accordingly, the token machine also checks this special case by comparing the thread ID received with the one held in it. If both values are equal, the token machine interrupts its internal context switch process immediately, abandons the AMTI and PC received and resumes fetching bytecode from the previous position again.

4.6.4 Implementation of Java Monitor Construct

As described in Section 2.3.5, a monitor inside the scheduler is mapped to a fat lock held in the monitor table. In this section, we first provide an overview on the data structure of a fat lock. Then, we introduce how fat lock slots in the monitor table are efficiently managed at runtime. After that, we discuss the implementation of the monitor operations in detail. At last, we describe how the priority inheritance protocol is realized based on the thread and fat lock structures.

Fat Lock Structure

Similar to the thread structure, the data structure of a fat lock also consists of 14 attributes, which are described in Table 4.29 briefly, where $n_m = log_2 N_m$ and N_m represents the maximum number of monitors allowed to be used simultaneously.

Note that the thread and fat lock structures contain four identical attributes, namely `LOCK_COUNT`, `MONITOR_PRIORITY`, `MONITOR_PREVIOUS` and `MONITOR_NEXT`. As a thread invokes the **`wait`**-method on an object to release its monitor, the current values of the attributes of the monitor are copied from the monitor table to the thread table. Once the thread reenters the monitor later, these values are copied back, which resets the monitor to the status before calling the **`wait`**-method. We refer to these four attributes as context attributes in this section below.

One important implication of Table 4.29 is that the maximum number of alive threads that may be used concurrently is currently limited to 64. Thus, the ID of the owner thread can be represented with 6 bits. The major reason for this limitation is that 64 threads should cover the vast majority of applications in the field of embedded systems. Also, this simplifies the fat lock structure so that an entire thread queue can be held in only two attribute values, which allows the thread queue to be accessed in a single clock cycle through the dual ports of the monitor table. If some customized version of the AMIDAR processor

No.	Attribute	Width	Description
0	HANDLE	32-bit	The reference to the object of the monitor in integer form.
1	OWNER	6-bit	The ID of the owner thread.
2	LOCK_COUNT	10-bit	The recursive lock count of the monitor.
3	MONITOR_PRIORITY	10-bit	The one-hot priority of the monitor.
4	MONITOR_PREVIOUS	n_m-bit	The previous monitor in the monitor chain.
5	MONITOR_NEXT	n_m-bit	The next monitor in the monitor chain.
6	BLOCKED_TQ_L	32-bit	The low part of the blocked thread queue.
7	BLOCKED_TQ_H	32-bit	The high part of the blocked thread queue.
8	BLOCKED_TQM_L	32-bit	The low part of the mask for the blocked thread queue.
9	BLOCKED_TQM_H	32-bit	The high part of the mask for the blocked thread queue.
10	WAITING_TQ_L	32-bit	The low part of the waiting thread queue.
11	WAITING_TQ_H	32-bit	The high part of the waiting thread queue.
12	WAITING_TQM_L	32-bit	The low part of the mask for the waiting thread queue.
13	WAITING_TQM_H	32-bit	The high part of the mask for the waiting thread queue.

Table 4.29: Monitor attributes

requires less than 32 threads, the high parts of the blocked and waiting thread queues as well as their masks are eliminated automatically.

Management of Fat Lock Slots

One of the key design goals of the lock model used in the AMIDAR processor is that only an active monitor (i.e. at least a thread is synchronized on the object of the monitor) may have a slot in the monitor table. This slot must be able to be addressed efficiently through the reference to the object to increase the performance of executing synchronization-specific operations. Once the object is not synchronized, its slot must be freed immediately so that the slot can be reused by another monitor. Below, we first introduce two basic components adopted to facilitate managing fat lock slots and then describe a simple architecture that combines both components together to support up to 63 active monitors at the same time. After that, we discuss an advanced architecture that increases the maximum number of active monitors greatly.

- ***Free Monitor Queue:*** This is just an instance of the thread queue circuit described in Section 4.6.1. Each nonzero bit in the queue corresponds to an unused fat lock slot. Each time a monitor becomes active, the leftmost free slot is selected by using an arbiter. The output of the arbiter needs to be converted to a binary value that represents an index into the monitor table and is used as the ID of the monitor.

- ***CAM:*** A CAM maps the reference to a given object to its monitor ID, if the object is currently being synchronized (i.e. its monitor is active). Otherwise, a *miss* signal is asserted, which causes that the scheduler assigns a free slot to the monitor of the object. Then, the object reference is written in the location of the CAM, which is addressed by the newly assigned monitor ID. Once the monitor becomes inactive, a null reference (i.e. 0) is written in the same location to clear the old content.

The simple architecture used to manage fat lock slots includes only a single free monitor queue and a CAM. Both of them have the same size, namely 64. After system reboot, all bits in the queue are set to 1 except the least significant one. Monitor ID 0 is reserved to represent the end of a monitor chain. Therefore, the maximum number of active monitors is limited to 63. This limitation should be feasible in a broad variety of situations because even heavily multi-threaded applications typically do not require more than 40 active monitor at the same time, as noted in Section 2.3.5.

However, to ensure a high scalability of the scheduler, a sophisticated architecture has been designed. Without concern for resource and timing constrains, this architecture could allow arbitrarily many monitors to be used simultaneously. It includes multiple pairs of free monitor queues and CAMs of same size. For ease of representation, we refer to such a pair as a monitor set. Each monitor set is assigned an ID to distinguish it from other sets. The size of a monitor set needs to be 2^i, where $i \in [1, 6]$. When using this architecture, the ID of a monitor can be considered as a concatenation of a set ID and a relative monitor ID inside the set. For example, given an instance of this architecture that consists of 4 monitor sets of size 64, the monitor ID 255 can be represented in binary form as $\{11_2, 111111_2\}$. The high two bits correspond to monitor set ID 3, whereas the low 6 bits relative monitor ID 63. Just like in the simple architecture above, monitor ID 0 is reserved for the same purpose, which indicates that the 0th bit of the free monitor queue in monitor set 0 stays unset after system reboot.

As Figure 4.23 illustrates, free monitor queues and CAMs are contained in two individual pools separately. Figure 4.38 demonstrates the detailed structure of a monitor pool which includes 4 free monitor queues. As a free monitor ID is needed, the monitor pool outputs the nonempty free monitor queue with the largest ID. Then, a relative monitor ID can be selected from this queue. The final monitor ID results from the combination of the ID of the selected queue and the relative monitor ID. On the input side, if a monitor ID needs to be enqueued or dequeued, the monitor pool generates a select signal for each of the free monitor queues according to the set ID. In this way, the enqueue or dequeue operation is solely performed on the free monitor queue determined by the set ID.

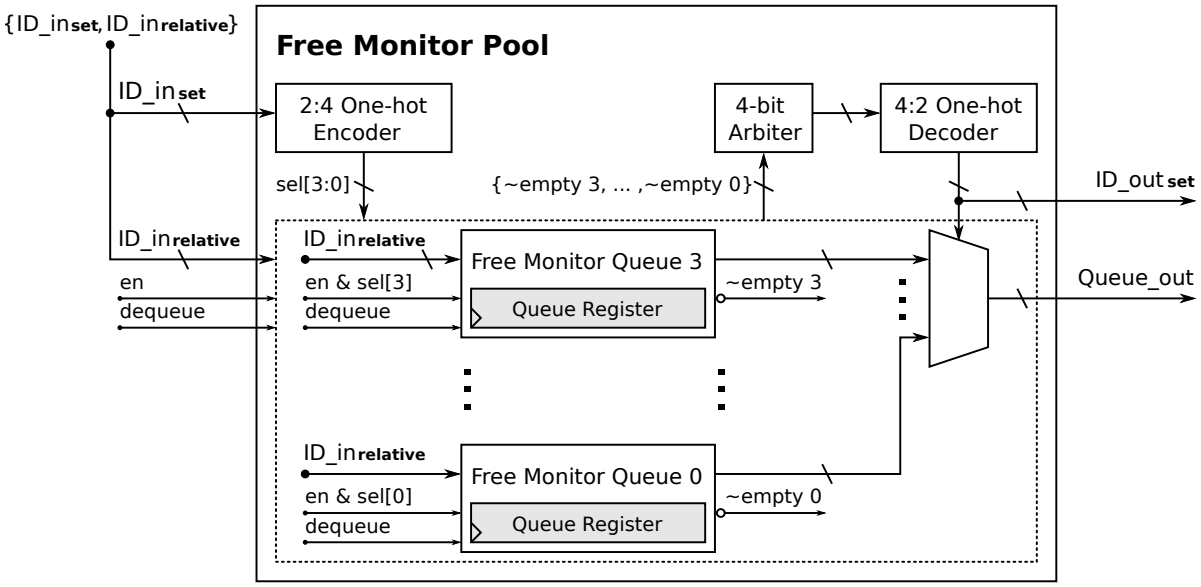


Figure 4.38: Free monitor pool

The structure of a CAM pool is similar to that of a free monitor pool. If an object reference needs to be written into the CAM pool through a monitor ID, the set ID contained in the monitor ID is only used to select a CAM, while the relative monitor ID locates a write position in the selected CAM. Later, upon converting this reference to the monitor ID of the object, the reference is broadcast to all CAMs. Only the CAM holding the reference asserts the *valid*-signal and outputs the relative monitor ID of the object. Using an one-hot decoder, the bit-vector consisting of the *valid*-signals of all CAMs is transformed to the corresponding set ID. Concatenating both IDs together results in the monitor ID of the object. If the reference of an object does not exist in the CAM pool, the *miss*-signals of all CAMs are set, causing that a free slot is assigned to the object.

Implementation of Thread Synchronization

Thread synchronization is based on the 6 operations listed in Table 4.26, namely `MONITORENTER`, `MONITOREXIT`, `WAIT`, `TIMED_WAIT`, `NOTIFY` and `NOTIFYALL`. Both `WAIT` and `TIMED_WAIT` are actually performed under the control of the same FSM, and so are `NOTIFY` and `NOTIFYALL`. All these operations may only be invoked by the currently running thread and thus do not need a thread ID to be explicitly passed in. In the following, we discuss the implementation of these operations in detail under the assumption that the monitor manager uses a single free monitor queue and a CAM to manage fat lock slots.

First, we describe the monitor locking and releasing processes by taking an object through a series of operations under different conditions.

Locking without Contention: Initially, object O is not synchronized and therefore no reference to it exists in the CAM. Thread T_0 tries to acquire its monitor through invoking a synchronized method on it. Upon the assertion of the *miss*-signal of the CAM, a free slot in the monitor table is assigned to the monitor of O. In the meantime, the reference to O is written into the CAM and the monitor table respectively. Then, the scheduler sets T_0 as the owner of the monitor and initializes the lock count of the monitor to 1. Note that the reference to a thread or monitor object is only used for the purpose of debugging. Inside the scheduler, a thread or monitor is referred to solely by its ID. For example, setting some thread as the owner of a monitor means that the value of attribute `OWNER` of the monitor is set to the ID of the thread. To support the priority inheritance protocol, a monitor chain needs to be established and maintained for each thread. Only the head and tail of a monitor chain are held in the thread table. Each monitor keeps its predecessor and successor in its fat lock structure. Assume that T_0 does not own any other monitor. Thus, both monitor chain head and tail of T_0 are assigned the monitor of O in the thread table. In the monitor table, the values of attributes `MONITOR_PREVIOUS` and `MONITOR_NEXT` of the monitor of O remains 0, indicating that this monitor does not have a predecessor or successor. A monitor is attached to the monitor chain of a thread, only as the thread enters it for the first time. Up to now, the monitor entering process is complete.

Nested Locking: Now assume that T_0 calls another synchronized method on O and thus needs to enter the monitor of O again. Through the monitor ID provided by the CAM, the lock count of the monitor is read out and compared with 0 to check whether the monitor is locked. Concurrently, the owner thread ID of the monitor is also read out and compared with the ID of T_0 by exploiting the second access port of the monitor table. Although the monitor is already locked (i.e. the lock count is greater than 0), but by T_0 itself, its lock count is therefore simply incremented by 1 and written back into the

monitor table, i.e. the thread succeeds to enter the monitor for the second time. A key thing to note is that the scheduler pushes the thread ID of T_0 into the next TID buffer in both cases described above to allow T_0 run further.

Locking with Contention: At some later time, T_0 is preempted by another thread, T_1, which also requires the monitor of O. To simplify the description, we assume that T_1 has the same priority as T_0. T_1 first checks if the required monitor is free by comparing the lock count with 0 and the owner thread ID with the ID of T_1 itself. Since T_0 has not released the monitor yet, the check fails. As a result, T_1 is moved from the ready thread queue to the blocked thread queue of the monitor. Its state is changed to *BLOCKED* and its needed monitor is set to the monitor of O. Also, the current priority of T_1 which is held in the priority table is compared with that of T_0. If T_1 has a greater priority than T_0, a priority inheriting process is triggered directly. Otherwise, the scheduler additionally checks whether T_0 has already inherited the priority of T_1. The reason for this additional check and the priority inheriting process are described in the next section below. Since both checks fail, the monitor entering process ends without further operation. Then, the scheduler switches the execution context back to T_0.

Unlocking: After T_0 is restarted, it returns from the second synchronized method invoked on O. Consequently, the lock count of the monitor is decremented by 1. Since the updated lock count is still greater than 0, the unlocking process terminates immediately. After a while, T_0 releases the monitor of O again on return from the first synchronized method, causing that the lock count of the monitor becomes 0, which indicates that the monitor is now free. As a result, several attributes of T_0 and the monitor need to be cleared, including the monitor chain head and tail of T_0, the owner and lock count of the monitor. After that, the scheduler attempts to select a new owner for the monitor. To do that, it first loads the mask of the blocked thread queue of the monitor into the shared thread queue circuit as described in Section 4.6.1. Since the mask does not contain nonzero bit (i.e. no scheduling round has been started yet), the blocked thread queue itself is then loaded into the thread queue circuit. This causes that T_1 is selected as the new owner of the monitor. Accordingly, T_1 is removed from the blocked thread queue of the monitor and added back to the ready thread queue. The updated blocked thread queue and the new mask provided by the shared WRRA are written into the monitor table in sequence. Note that the new mask still does not include any nonzero bit. Furthermore, the state of T_1 becomes *READY* and its needed monitor is cleared. The attributes of the monitor are updated in the same way that they were updated when T_0 locked the monitor for the first time.

Based on the example above, we explain below how the wait-notify mechanisms of Java is implemented in the scheduler. One necessary condition for a thread to invoke the **`wait`**-, **`notify`**- or **`notifyAll`**-methods on an object is that the thread has owned the monitor of the object.

Waiting: Assume that T_0 invokes the **`wait`**-method on O after T_1 has been blocked. First, T_0 is moved from the ready thread queue to the waiting thread queue of the monitor. It also needs to be added to the timed-waiting thread queue, if the **`wait`**-method has been invoked with a timeout value. The state of T_0 is changed to *WAITING* and its needed monitor is set to the monitor of O. After the current values of the four context attributes of the monitor, namely `LOCK_CNT`, `MONITOR_PRIORITY`, `MONITOR_PREVIOUS` and `MONITOR_NEXT` have been copied to the thread table, these attributes as well as `OWNER` of the monitor are all reset to 0 in the monitor table. At last, the scheduler attempts to select a thread from the

blocked thread queue of the monitor or its mask and then assigns it the monitor if one can be found. Consequently, T_1 becomes the new owner of the monitor.

Notifying: Before T_1 releases the monitor of O, it tries to wake up one waiting thread by calling the **`notify`**-method on O. This causes the scheduler to start selecting a thread from the waiting thread queue of the monitor or its mask to awaken. As a result, T_0 becomes runnable again and is moved back to the ready thread queue. If T_0 is timed-waiting, it needs to be dequeued from the timed-waiting thread queue additionally. Note that T_0 does not own the monitor yet after it has been awakened. If the **`notifAll`**- instead of **`notify`**-method is invoked, the scheduler first dequeues all waiting threads from the timed-waiting thread queue speculatively and then wakes up each of them sequentially.

Relocking: After T_0 is restarted, it first needs to lock the monitor of O again. Since it has a valid monitor context which is indicated by the nonzero value of `LOCK_CNT`, the scheduler simply copies the values of the four context attributes back to the monitor table and then clears them in the thread table.

Implementation of Priority Inheritance

In this section, we explain the implementation of the priority inheritance protocol. First, we define two different types of chains that are needed for the following discussion.

- ***Thread chain***: Multiple threads are tied up in a thread chain, if the predecessor of each thread in the chain requires a monitor owned by the thread and therefore blocks.
- ***Monitor chain***: This chain consists of all monitors that a thread is currently owning in a FIFO order. This indicates that the chain head corresponds to the first monitor that the thread has entered, while the chain tail the last one.

As described above, the thread structure contains an attribute called `PRIORITY`. Most of the time, it just holds the original priority of a thread in one-hot form until the thread inherits a higher priority. In this case, the one-hot value of the new priority is added to the current value of the attribute by performing a bitwise OR on them. Since a thread may inherit multiple priorities, we refer to the value held in this attribute as the *priority vector* of the thread. The effective priority of a thread which is stored in the priority table always corresponds to the highest priority in the priority vector and must be updated as soon as the priority vector varies.

To facilitate recovering the priority of a thread, the fat lock structure includes a corresponding attribute, namely `MONITOR_PRIORITY`, which usually just remains 0. However, if a monitor blocks some threads that have higher priorities than its owner, attribute `MONITOR_PRIORITY` of this monitor may also contain several priorities in one-hot form. Once a thread releases a monitor, all priorities held in this attribute are eliminated from the priority vector of the thread by performing a bitwise XOR on both values. In the rest of this section, different priority inheritance scenarios are described, from simple to complex.

Priority inheriting without updating thread chain: Initially, thread T_0 with priority 5 owns a single monitor, M_0, which does not block any thread. This means that the priority vectors of T_0 and M_0 are $00\_0001\_0000_2$ and 00_0000_0000 respectively. Later, two threads with priority 7 and 8 attempt to enter M_0 in sequence and therefore block, causing that the priority vectors of T_0 and M_0 are changed to $00\_1101\_0000_2$ and 00_1100_0000. Accordingly, the effective priority of T_0 is updated to 8. Then, T_0

enters a second monitor M_1 before a thread with priority 10 that also requires the monitor. Consequently, the priority vectors of T_0 and M_1 become $10\_1101\_0000_2$ and $10\_0000\_0000_2$. As a result, the priority of T_0 is raised to 10. Once T_0 releases M_1, its priority vector is changed back to $00\_1101\_0000_2$ and its effective priority decreases to 8. The value of attribute `MONITOR_PRIORITY` of a monitor is cleared as the monitor is released. Finally, T_0 frees M_0, which causes that its priority falls back to 5 and its priority vector contains solely this priority in one-hot form.

Priority inheriting with updating thread chain: A key thing to note is that priority inheritance is transitive [96]. Assume that threads T_0, T_1 and T_2 have priorities 5, 7, 8 and own monitors M_0, M_1, M_2 respectively. They form a thread chain in such a way that T_0 requires M_1 and T_1 requires M_2, i.e. T_0 is the chain head and T_2 the chain tail. Since these threads are ordered in descending order of priority, none of them inherits another priority. This means that the priority vectors of T_0, T_1 and T_2 only contain their original priorities and the priority vectors of three monitors are all 0. Inside the scheduler, these threads are linked together through attribute `NEEDED_MONITOR`. At some later time, thread T_3 with priority 10 fails to enter M_0 and blocks. This has the consequence that all of T_0, T_1 and T_2 inherit priority 10 due to the transitiveness of priority inheritance. Accordingly, their priority vectors become $10\_0001\_0000_2$, $10\_0100\_0000_2$ and $10\_1000\_0000_2$, while the priority vectors of all three monitors are changed to $10\_0000\_0000_2$. To achieve this, the scheduler checks additionally the state of a thread after this thread has inherited a higher priority. If the thread is blocked and the priority of the owner of its needed monitor is less than the inherited one, that owner thread will also inherit the same priority. The priority inheriting process is repeated until one thread in the chain has a higher priority or the chain tail is reached.

Updating monitor chain without inheriting priority: The monitor chain of a thread needs to be updated in a special case as described below. Assume that thread T_0 with priority 5 enters M_0, M_1 and M_2 sequentially, which forms a monitor chain whose head is M_0 and tail M_2. Later, thread T_1 with priority 7 tries to acquire M_2 and fails, which changes the priority vectors of T_0 and M_2 to 00_0101_0000 and 00_0100_0000. Then, another thread T_2 that also has priority 7 is blocked by T_0 due to M_0. In this case, the priority vectors of M_0 and M_2 need to be updated to 00_0100_0000 and 00_0000_0000 respectively. Consequently, the priority of T_0 remains 7 until M_0 becomes free. Otherwise, its priority would already fall back to 5 after T_0 had released M_2, while T_2 was still blocked, which would cause the priority inversion problem in a latent way. To avoid this, each time a thread blocks another thread that has a lower priority than it or the same priority as it, the scheduler additionally checks if the thread has inherited the priority of the blocked thread before. If this is the case, the scheduler traverses the monitor chain of the thread to find the appropriate monitor to inherit the higher priority. This process can be formally described as such: *Assume that the monitor chain of a thread T consists of M_0, ..., M_n, where the subscript of a monitor represents the order in which the monitor is entered by T. If multiple threads with the same higher priority, namely p, are blocked by different monitors of T, only the monitor with the smallest subscript may inherit p.* In this way, p is removed from the priority vector of T only after the last monitor that blocks at least a thread with priority p has been released.

4.6.5 Interrupt Handling

As mentioned above, the AMIDAR processor uses a wait-interrupt-based interrupt handling model that has been integrated into the thread scheduling framework completely. The major feature of this model

is that an incoming interrupt request (IRQ) is considered to be valid only if the corresponding interrupt service thread (IST) is just waiting for its arrival. This allows programmers to handle external events that occur randomly in an interactive way, resulting in more elegant and structural codes. In this section, we describe the implementation of this model at both software and hardware levels.

Interrupt Service Thread

At the software level, each interrupt source is assigned a dedicated IST. For the AMIDAR processor, an interrupt source can be either a peripheral device like UART or an FU like the heap manager. ISTs are instantiated by the bootloader after system reboot and before invoking the **main**-method. The instantiating order of ISTs corresponds to the order in which their interrupt sources are connected to the AMIDAR processor in the system builder and therefore can be changed arbitrarily. A key thing to note is that interrupt sources are prioritized according to their instantiating order, i.e. the first interrupt source connected to the AMIDAR processor in the system builder has the highest priority among all interrupt sources and the last one the lowest priority. The priorities assigned to interrupt sources are independent of the 10 standard thread priorities defined in Java. They are solely adopted to facilitate selecting an IST when multiple IRQs are asserted simultaneously.

Every interrupt source has a corresponding Java class which describes its Wishbone-interface. Each field of the class represents one of the Wishbone-registers of the interrupt source. The class **de.amidar.AmidarSystem** includes a static field called **ISTPool**, which refers to a hash map. During booting, a newly created IST object is added to **ISTPool** by using the class of the corresponding interrupt source as key. The IST is neither started nor initialized by the bootloader. Therefore, a programmer can initialize the attributes of an IST as desired, define the concrete task (i.e. ISR) run by it using a **Runnable**-object and then starts it at an appropriate time, providing the maximum flexibility to program design and development.

Listing 19 illustrates the **run**-method of class **de.armidar.UartHandler** that implements the interface **Runnable**. It defines the ISR of the UART peripheral used by the AMIDAR processor. Note that this method is a simplified version of the original one and only contains the part handling data transmission. Before the IST of the UART is started, an object of **UartHandler** needs to be generated and assigned to the field **target** of the IST through invoking the **setTask**-method of class **Thread**. In such a way, the instantiation of an IST is decoupled from the definition of its task completely. It would be even possible to replace the task executed by an IST at runtime.

As the listing demonstrates, the task of the IST is defined inside an endless loop, allowing the IST to handle IRQs from UART throughout the entire lifetime of an application. An IST should be set as daemon so that it terminates automatically after all user threads have terminated. After the IST of the UART is started, it first checks if the buffer **uartFifo** holds data written by user threads. This buffer can be considered as the communication medium between the IST and user threads. A user thread plays the role of a producer and writes data to the buffer, while the IST reads data from it as a consumer. Since the buffer is shared by multiple threads, it needs to be accessed in a critical section to avoid race conditions. Once the buffer becomes full, the writing user thread signals the IST via the invocation of the **notify**-method on the buffer and then starts waiting. The awakened IST transmits data held in the buffer until it becomes empty or the input buffer of the UART is full. Then, it wakes up the waiting user

Listing 19: UartHandler.run()

```
 1: int maxBytes;
 2: while(true){
 3:     // interface between the IST and user threads
 4:     synchronized(uartFifo){
 5:         while (uartFifo.isEmpty()) {
 6:             try {
 7:                 uartFifo.wait();
 8:             } catch (InterruptedException e) {
 9:                 e.printStackTrace();
10:             }
11:         }
12:         // AmidarSystem.invokeScheduler(DISABLE_CS ,0 ,0 ,0);
13:         maxBytes=uartFifo.contentSize()<16?uartFifo.contentSize():16;
14:         while(maxBytes-->0)
15:             uartPeri.push(uartFifo.pop());
16:         uartFifo.notify();
17:         // AmidarSystem.invokeScheduler(ENABLE_CS ,0 ,0 ,0);
18:     }
19:     // interface between the IST and UART
20:     synchronized (uartPeri) {
21:         try {
22:             uartPeri.wait();
23:             uartPeri.clearInterrupt();
24:         } catch (InterruptedException e) {
25:             e.printStackTrace();
26:         }
27:     }
28: }
```

Listing 20: Simplified version of the waitObject-method

```
 1: // convert obj to an integer
 2: int o2i = AmidarSystem.intToRef(obj);
 3: // start waiting on obj
 4: AmidarSystem.invokeBlkScheduler(WAIT,o2i,0,0);
 5: // wake up an IST explicitly, if obj is an interrupt source
 6: if(obj instanceof AmidarPeripheral) {
 7:   AmidarSystem.invokeScheduler(NOTIFY,o2i,0,0);
 8: }
 9: // reenter the monitor of obj before return
10: AmidarSystem.invokeBlkScheduler(MONITORENTER,o2i,0,0);
```

thread and starts waiting on the object **uartPeri**. This object is an instance of the class of the UART, through which the Wishbone-registers of the UART can be accessed.

After the transmission is complete, the UART asserts a level-sensitive IRQ signal, causing that its IST is restarted by the scheduler. However, the whole interrupt handling process as described below is performed asynchronously to the execution of the Java program at the hardware level. To synchronize both software and hardware levels, the IST needs to be awakened explicitly by exploiting the FU-NI of the thread scheduler before return from the **wait**-method. Currently, the **wait**-method is implemented based on the **waitObject**-method of class **de.amidar.Scheduler**. Listing 20 shows a simplified version of this method that does not handle timeout values, where **obj** corresponds to an object on which the **wait**-method is invoked. As the listing demonstrates, once an awakened thread is restarted, it first checks whether **obj** is an interrupt source or a normal object (line 6). In the former case, it invokes the NOTIFY operation provided by the thread scheduler on **obj** to wake up itself (line 7). Furthermore, as mentioned in Section 4.6.4, an awakened thread must reenter the monitor of **obj** before it may leave the **wait**-method and run further (line 10). After return from the **wait**-method, the IST clears the IRQ by invoking the **clearInterrupt**-method as shown in Listing 19 at line 23. Then, the IST starts transmitting data again, since it does not need to handle data reception.

There are three important things to note about the interrupt handling model of the AMIDAR processor. First, the IRQ signals of all interrupt sources must be level-sensitive like that of the UART. Second, the interrupt handler contained in the scheduler is deactivated after an IRQ is detected, and is reactivated when the interrupt source ceases asserting the IRQ. Last, after an IST has been started, it enjoys no privilege over normal user threads. The major advantage of this model is to allow interrupts to be handled completely under the thread scheduling framework. Through the use of the wait-interrupt-based interactive mechanism between an IST and its interrupt source, the occurrence of an interrupt is not asynchronous anymore, providing programmers more control over random external events at the software level. Also, this model can be easily extended to fit into the classical interrupt handling schemes by modifying ISTs slightly. For example, if the invocation of the **clearInterrupt**-method in Listing 19 was removed from line 23 and inserted before both line 7 and 22, the IST could not be interrupted by another incoming IRQ during executing the main part of the ISR, which would result in a classical non-nested interrupt handling scheme. Furthermore, to avoid the data transmission (line 13-16) to be preempted by another user thread, the context switch process could be temporarily deactivated (line 12) and then reactivated (line 17) after the transmission was complete.

Interrupt Handling inside the Scheduler

The major tasks for handling interrupts at the hardware level include detecting the occurrence of valid IRQs, selecting an appropriate IST as well as context switching. The former two are performed by the interrupt selector, while the latter one by the interrupt handler.

As Figure 4.23 illustrates, both of the IRQ bus and the waiting IST queue are connected to the interrupt selector as input signals. The lines held in the IRQ bus are ordered by the system builder automatically according to the order in which the corresponding interrupt sources are connected to the AMIDAR processor. As mentioned above, the ISTs of these interrupt sources are instantiated by the bootloader in the same order. Additionally, since ISTs are the first threads created in a program, they are

assigned the largest thread IDs allowed in an AMIDAR system[19]. This means that only the highest N_{IRQ} bits in the waiting IST queue are employed to represent ISTs and can actually be set to 1, where N_{IRQ} is the width of the IRQ bus (i.e. the number of the interrupt sources). Note that these N_{IRQ} bits can be mapped directly to the lines in the IRQ bus one by one due to the way how they are ordered.

For example, assume that a customized AMIDAR system consists of 3 interrupt sources and a processor with support for up to 8 alive threads. The 3 ISTs are assigned ID 7, 6, 5 and the IRQ signals of their interrupt sources are connected to the processor via line 2, 1, 0 of the IRQ bus respectively. Due to this 1:1 mapping relationship, the IRQ bus is used to mask the highest N_{IRQ} bits of the waiting IST queue inside the interrupt selector. All nonzero bits remaining in the masked waiting IST queue are valid IRQs that need to be handled. Therefore, the occurrence of valid IRQs can be identified by simply performing an OR-reduction on the highest N_{IRQ} bits of the masked waiting IST queue.

Since multiple IRQs can be asserted at the same time, the interrupt selector needs to select one from them to handle, which can be achieved through two different strategies: either according to their priorities or in a round-robin manner. To realize the former strategy, the masked waiting IST queue can be input into an arbiter described in Section 4.6.1, which ensures that the highest-priority IRQ is always preferred. The round-robin-based IST selecting strategy can be easily implemented by replacing the arbiter with a RRA. After an IST has been determined, its thread ID in one-hot form is converted to the binary format by an one-hot decoder. Thus, regardless of which IST selecting strategy is adopted, it always takes 2 clock cycles from the occurrence of valid IRQs to outputting the thread ID of the IST that needs to be stared.

As soon as the IRQ_valid-signal becomes active, the interrupt handler requests a context switch. Since the context switcher favors requests from the interrupt handler over requests from the token machine and system timer to reduce interrupt latency, an IST can be considered as a thread with a priority level of infinity before it is started. Once the requested context switch is complete, the interrupt handler starts waiting for resetting the corresponding IRQ at the software level and does not react to another IRQ until the current one is cleared.

4.7 AMIDAR Debugging Framework

In the following, the AMIDAR debugging framework is described in detail. First, Section 4.7.1 explains its working principle and implementation. In the next section, several use cases are discussed. Finally, its performance and resource usage are presented in Section 4.7.3.

4.7.1 Concept and Implementation

The AMIDAR debugging framework provides hardware and software debugging functionalities for embedded systems implemented on an FPGA. Both features work over a single JTAG connection and complement one another to form a powerful debugging tool for soft-core processors. Figure 4.39 illustrates the concept of the framework. The system is composed of the three entities *debugger*, *protocol converter* and *target*. Components filled with color are provided by other libraries and tools, white ones are part of the newly developed framework. Eclipse serves as user interface for the debugger components. However, most of the components are implemented independently of the Eclipse platform in a standalone library.

[19] The main thread is started by default after system reboot and has ID 0.

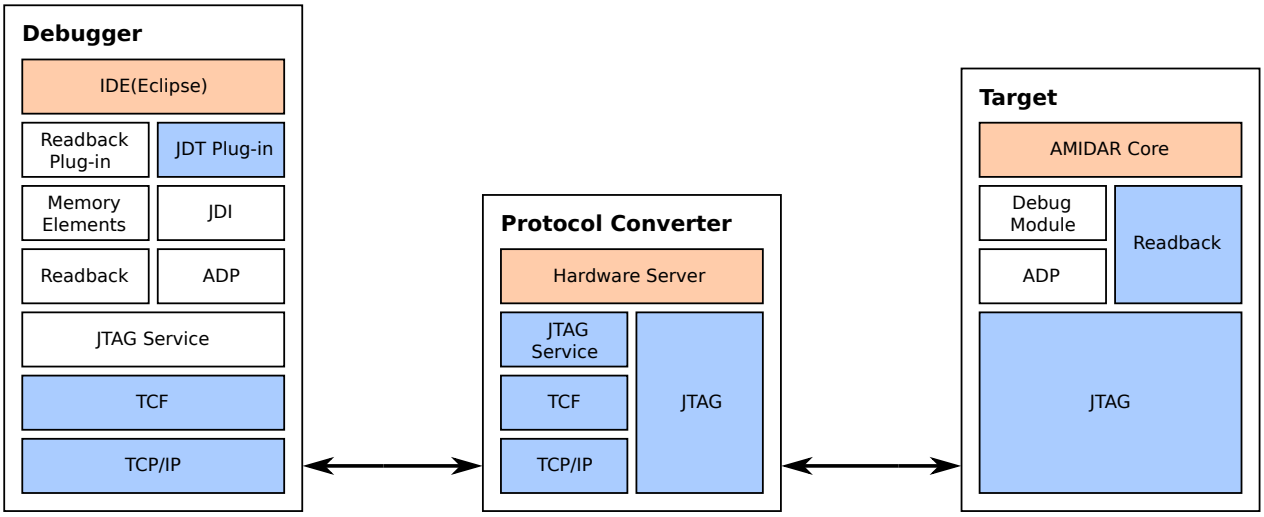


Figure 4.39: Concept of the AMIDAR debugging framework

In order to allow debugging a Java program which runs on the AMIDAR processor just like one which runs on a JVM, a custom implementation of the *Java debug interface* (JDI) [48] is part of the framework. The *Java development tools* (JDT) plug-in of the *Eclipse* platform relies on this interface for debugging normal Java applications. By providing the custom JDI implementation, existing Eclipse plug-ins do not need to be modified. This JDI implementation makes use of the *AMIDAR debug protocol* (ADP), which is a special protocol built upon a JTAG connection, to communicate with the hardware target. The JTAG connection is provided by *Xilinx hardware server*. It implements all OS and vendor specific mechanisms for accessing JTAG adapters. Clients can connect to this server using the *target control framework* (TCF), which relies on TCP and is publicly available as Eclipse plug-in or as standalone Java library [106]. The debugging framework only needs to implement the JTAG extension of the TCF. Consequently, all software components of the debugger are written platform independently in Java. The AMIDAR core on the target FPGA contains a debugger FU that decodes the commands received over the JTAG port and controls the core accordingly.

Hardware debugging functionality can be used independently from software debugging functionality. It uses the same JTAG connection to the target and requires almost no additional hardware components. The readback component of the framework reads the configuration memory of the FPGA, which also contains values of memory elements being part of the FPGA design. These values are extracted and recombined according to the memory elements found in the HDL source code. A special Eclipse plug-in displays the values to the user. Additional commands of the debugger FU allow the clock signal of the processor core and its peripherals to be controlled.

As an alternative to the Eclipse IDE, most features can also be accessed by means of a simple command line interface. Furthermore, the framework can be utilized in automatic tests. This enables an easy implementation of tests for hardware and software components of a real system.

Hardware Components

For software debugging, some basic features were added to the token machine of the AMIDAR processor, as described below:

- It can be halted and resumed by an external signal.
- A software breakpoint mechanism has been integrated. A special breakpoint bytecode is placed at every position in the code memory where the token machine should stop execution automatically.
- The token machine can be configured at runtime to halt on exceptions.
- Stepping can be performed for a single bytecode or up to a certain program counter value.

Whenever the token machine is stopped by any of the mentioned hardware events, a status signal indicates the type of event. With these mechanisms, the debugger FU can control execution of the Java program on the processor. Active threads are indicated by a simple connection from the thread scheduler to the debugger FU. Hence, thread states can be determined at any time without influencing the processor execution.

All other software debugging features are realized by universal interfaces to the token distribution network and the data bus of the AMIDAR processor. While the token machine is suspended, arbitrary tokens and data packets can be sent to the FUs of the processor by the debugging framework via the debugger FU. To read an object field for example, the corresponding token, handle, and offset are sent to the heap manager which returns the field value. Consequently, the debugger uses the same access to the data as the program being executed on the processor and does not need to care about hardware implementation details like caching. Hence, the AMIDAR model simplifies the implementation of debugging features. A similar principle has already been applied by previous ARM architectures [9]. They allow instructions to be scanned into the core during debug mode. Nevertheless, our approach allows a finer access to the AMIDAR core by addressing its components separately. This permits debugger actions which are difficult to achieve by processor instructions. For example, FUs can be tested individually in real hardware. Since the heap manager provides access to arbitrary addresses of the external DRAM, this universal interface also allows to place breakpoints in the code memory.

Since the hardware debugging functionality relies on the readback mechanism, which is integrated into modern FPGAs by default, almost no additional hardware components are necessary for this feature. The debugger FU allows the clock signal of the AMIDAR core and its peripherals to be stopped and stepped by commands received over the JTAG interface. This is realized by controllable clock buffers as proposed in [47].

Java Debugging

Java debugging features are provided by a custom implementation of the JDI. Since this interface is designed for JVMs, it is very extensive and powerful. Not all of its features have been implemented yet. Nevertheless, all core functionalities are available. Breakpoints can be added or removed at any time. The processor can be stopped by exceptions but a distinction between caught and uncaught exceptions is not possible. Stepping is supported in all modes which are specified by the JDI. Thread switches are suppressed during step execution. Start and termination of Java threads are reported but the processor cannot be automatically stopped at these events. All JDI events only support stopping the whole processor, not a single thread. The call stacks of all Java threads can be retrieved. Reading and modifying fields or local variables are currently only possible while the processor is stopped. The mentioned limitations do not severely affect the debugging process of embedded systems.

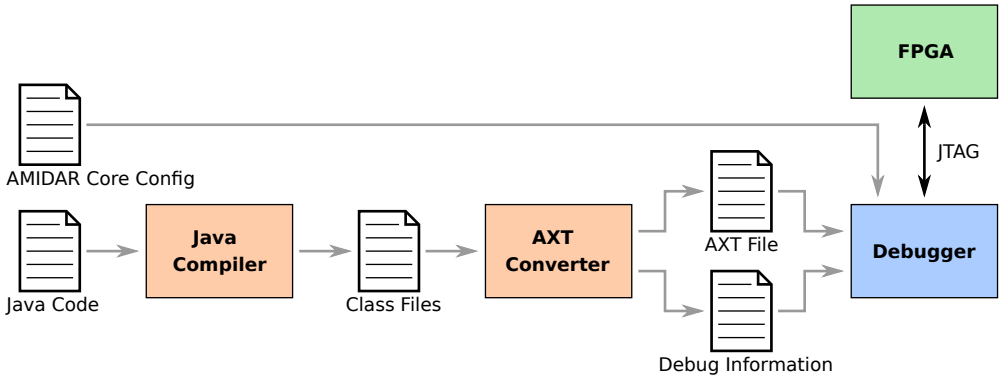


Figure 4.40: Tool flow for Java debugging

The corresponding tool flow is shown in Figure 4.40. Java source code is compiled by a standard Java compiler. Resulting binary class files are processed by a converter which generates an AXT file and a debug file. As described in Section 4.2, the AXT file contains only the binary code and data required for execution of the Java program. The debug file combines meta-information found directly in the class files like method names, variable names, or line number attributes. Furthermore, it links this information to the binary identifiers of the AXT file. This debug information enables the custom JDI implementation to provide the same abstract view onto the Java program as if it was executed on a JVM.

Software debugging features do not use the readback mechanism of the FPGA. One reason is that the major part of the software memory resides in the external DRAM which is not accessible by readback. Another reason is to avoid synchronization problems which arise from caches in the AMIDAR core. Instead of this, queries of the software state are translated into tokens and operands which are injected into the core by the debugger FU. This translation requires structural knowledge about the AMIDAR core, which is taken from the core's configuration file. Queues for tokens and operands exist on both software and hardware sides to enhance the throughput of the JTAG connection.

An important part of the JDI implementation is the event system. A dedicated thread of the framework repeatedly polls the state and event flags provided by the debugger FU. It generates appropriate JDI events when the state of the AMIDAR core or of a thread has changed. The debugger reacts to these events by reading out variables and the call stacks of halted threads. The mapping from hardware events to JDI events is not trivial because a single hardware event can trigger multiple successive JDI events. The *VM start event* and *class prepare events* are sent when the processor has finished executing the initialization routines and entered the main method. A breakpoint is used to detect this point in time.

Hardware Debugging

Figure 4.41 depicts the tool flow for hardware debugging functionality. It is based on the readback mechanism of Xilinx 7 Series FPGAs but could also work with similar mechanisms of other FPGAs. Information about the memory elements is extracted from the logic location file which can be generated by the `write_bitstream` command of Vivado. It contains the positions of all used memory elements in the configuration bitstream. Registers are identified by the name of the nets they drive, RAM elements are identified by their position on the FPGA. After the logic location file has been written, a TCL script is executed which exports all additional information required for reconstructing the values of memory elements according to the HDL source code. The first step is to replace the net names in the logic location

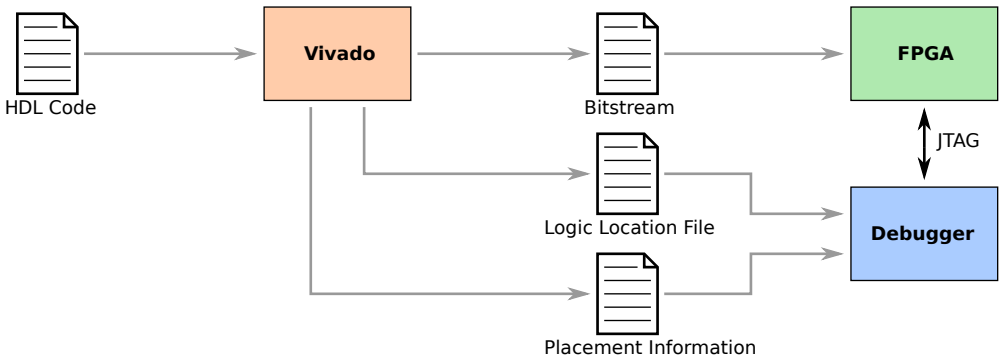


Figure 4.41: Tool flow for hardware debugging

file by the names of the driving registers because these are much more predictable after synthesis and in most cases match the corresponding names in the source code. All BRAM entries are removed except for the first bit of each block. The positions of BRAM bits in the bitstream relative to the first bit of each block are identical for all models of the 7 Series FPGA family. This makes it possible to reduce the size of the logic location file drastically, which also decreases the time for parsing it. The second step creates a file with placement information of RAM elements in the design. For each primitive RAM element, it contains its type, its position on the FPGA, its name and some additional attributes. Together with the logic location file, the debugging framework is able to create a data base which maps individual bits of the configuration bitstream to their positions in memory elements of the HDL source code.

While the FPGA is operating, the user can select memory elements of interest and start the readback procedure. Configuration frames which contain bits being part of a selected memory element are read out via JTAG. These bits are then extracted and recombined to the values of the memory elements.

Reading out BRAMs is only possible while they are not accessed by the FPGA design because of limited port resources. Other works about readback functionality use clock gating to freeze the FPGA design during readback access [8, 56]. However, this turns out to be problematic with complex designs consisting of multiple clock domains, especially in conjunction with an external DRAM. The DRAM controller cannot be frozen without data loss. As an alternative, we take advantage of the software debugging functionality in order to put the AMIDAR core into a safe state where no BRAM accesses are executed. Other IP cores like the DRAM controller can remain running. After the readback procedure the AMIDAR core can continue execution without any problems.

Although the recombination of configuration bits to memory elements according to the HDL source code works perfectly in most cases, it is not always possible. Some registers might be removed or the encoding of states might be changed during synthesis. Furthermore, the tool does not recognize all special mapping techniques yet. For example, the synthesis tool might use the second port of a BRAM to double the bit width of the memory.

Modifying the contents of memory elements would in principle be possible. But the transfer of bits between configuration memory and registers of the FPGA design can only be executed for the whole FPGA at once. Since our design cannot be frozen completely because of the DRAM controller, this mechanism cannot be applied in this case.

Eclipse Plugin

An Eclipse plug-in has been developed for convenient usage of the debugging framework. Integration of the Java debugging functionality required little effort because of the custom JDI implementation. Only the launching process of the software had to be realized. An Eclipse view for reading hardware memory elements is available. It lists the elements in a tree structure according to the design hierarchy. Elements of interest can be read manually at any time or automatically whenever the processor has been suspended.

4.7.2 Use Cases

This section discusses several typical use cases for the AMIDAR debugging framework from two different perspectives, according to the development of a simple distance meter.

Test Application

The distance meter includes an ultrasonic sensor and an OLED display with I2C and SPI interface respectively. These are connected to a Nexys Video board [75] with the Artix-7 FPGA from Xilinx, complete with an AMIDAR-based SoC. The key development goal is to write an application consisting of the drivers for I2C and SPI bus as well as a central control software. The control software reads current distance values from the ultrasonic sensor, recognizes and removes outliers using a median filter, and then shows the adjusted values on the display in a clean and pretty way.

Debugging Software

An application developer assumes that all of the hardware components (e.g. the sensor and the corresponding I2C master module) have already been tested previously and should work as designed and he only needs to concentrate on writing and debugging the software.

Manual setting of breakpoints: After the I2C driver has been completed, an unexpected bug occurs, namely, the driver always reads a constant value (e.g. 255) back from the sensor instead of the actual distance. The developer suspects that the bug should be in the driver just written, because, as mentioned above, all hardware components should have been tested thoroughly. This bug can be traditionally located by using either a simulator like ModelSim [72] or a logic analyzer like Vivado Debug Core [117]. One critical, but often overlooked aspect of debugging a soft-core based system is that an application developer typically does not get used to debug a software by observing waveforms of a number of signals. The graphical interface of the AMIDAR debugging framework simplifies fixing bugs like the one above. The developer only needs to insert several breakpoints at suspicious locations in the I2C driver and restart the system. Once the control flow reaches a breakpoint, the AMIDAR core will be suspended and the current values of all Java variables such as primitive object fields, strings or arrays will be represented in the most familiar way for a software engineer, as shown in Figure 4.42.

Exception based breakpoints: After the I2C driver has been capable of yielding correct distance values, a median filter is developed, which analyzes the last 30 values held in an array to eliminate outliers. However, an `UnsupportedOperationException` is thrown by the `arraycopy`-method during the online test. To find out the reason, the support for exception based breakpoints as shown in Figure

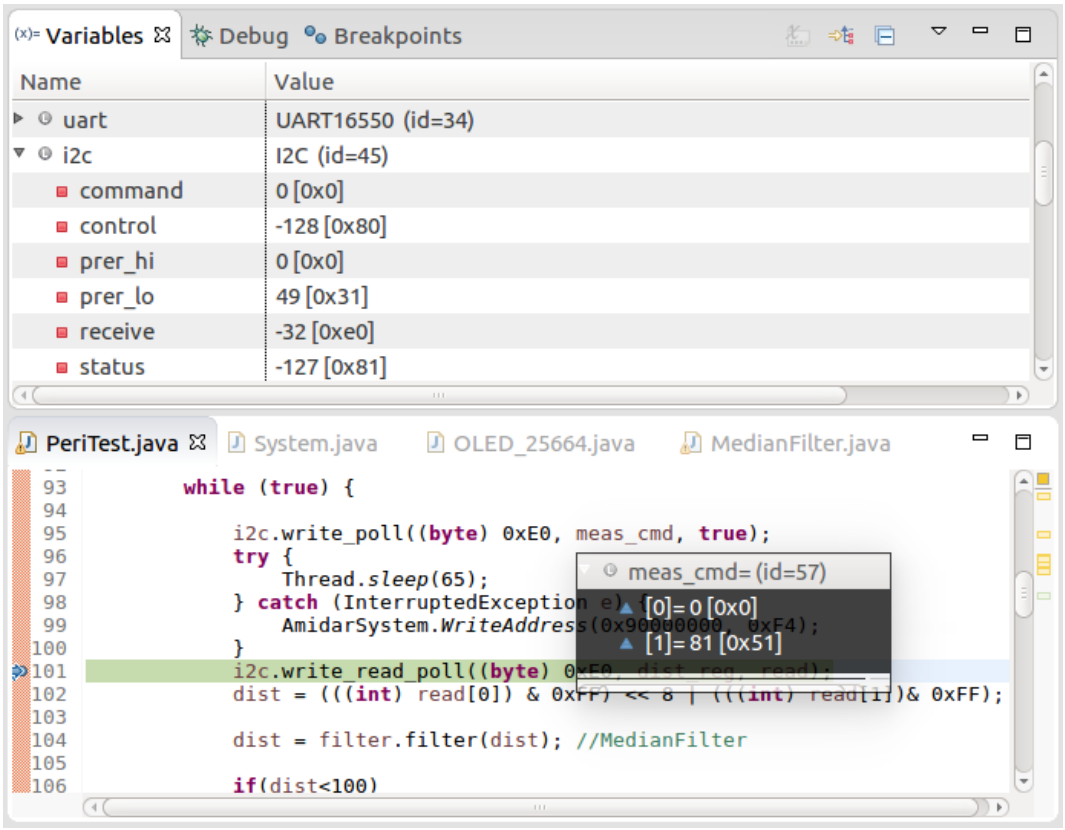


Figure 4.42: Debugging the software in Eclipse

4.43 is activated, which suspends the processor if any exception is thrown, allowing the developer to inspect the entire context of the exception accurately.

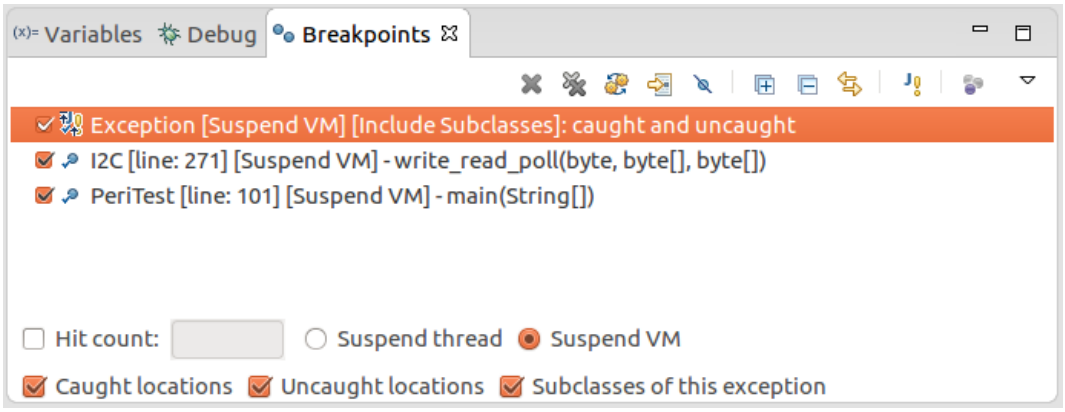


Figure 4.43: Activating exception breakpoints

Inspection of deadlocks: Sometimes, the execution of a software remains frozen for different reasons. For example, the median filter of the distance meter locks the array holding the distance values during the filtering and does not free it properly. Therefore, the I2C driver is not allowed to write new values into the array any more. In this case, the software execution can be easily paused by sending a `HALT` command to the debugger FU. The context in which the deadlock has occurred remains unchanged so that the programmer can find out the cause of this deadlock.

Debugging Hardware Modules

The distance meter works properly, after its software part has been debugged and tested by the application developer. However, the SPI driver transfers data to the display serially and therefore becomes the bottleneck of the system, especially when some complicated graphic needs to be refreshed on the fly according to distance values. Consequently, a hardware developer starts optimizing the SPI master module to enable DMA transmissions. Once he has finished all necessary local module tests on the SPI master, using ModelSim, he must test it in the context of the entire system further. To meet this goal, he can exploit the AMIDAR hardware debugger instead of the simulator.

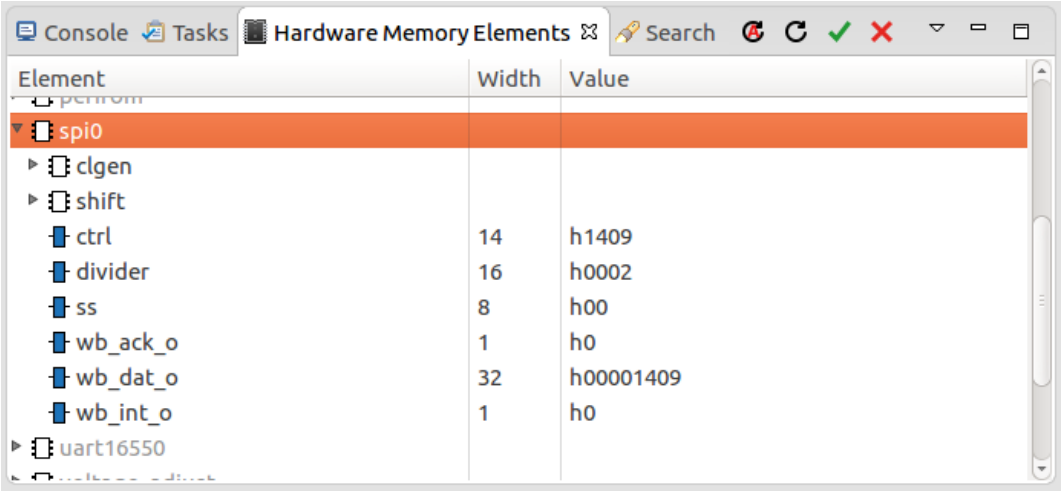


Figure 4.44: Inspecting hardware memory elements

Cycle accurate hardware inspection: The hardware developer first needs to write a suite of Java unit tests and run them on the distance meter. If all tests have been passed, he may add this new SPI master module to the peripheral kit of the AMIDAR processor directly. Otherwise, he should insert a breakpoint just before the DMA transmission, restart the failed tests and then step through all clock cycles needed by the transmission to monitor the detailed state of the SPI master as well as the DMA controller at each time step, as shown in Figure 4.44.

Profiling: The readback ability makes it very straightforward to collect profile information at the slight expense of hardware resource. Profiling counters can be integrated into an arbitrary hardware component such as the DMA controller. Such a counter will be incremented, only if some special event occurs, like starting a new DMA transmission. There is no need for any output logic because of readback. The profile information can be extracted from these counters at any time over the readback port.

4.7.3 Performance and Resource Usage

The AMIDAR debugging framework has been tested and evaluated with the SoC included in the distance meter described above. This SoC comprises about 45k look-up tables, 33k registers and 182 BRAMs. The debugger FU occupies 712 (1.6%) look-up tables and 1340 (4.1%) registers. Thus, hardware requirements are not negligible. However, 292 (0.7%) look-up tables and 975 (3.0%) registers are used for FIFOs to buffer the connection between the AMIDAR data bus and the JTAG port. Their sizes can be reduced at the expense of throughput. Furthermore, these requirements do not increase when additional FUs or peripherals are added to the system. Consequently, hardware requirements for providing both

mighty hardware and software debugging functionalities are low due to exploitation of the readback mechanism. Execution speed of the processor is currently not influenced by the debugging features.

The Eclipse instance including the debugging framework has been executed on an Intel Core i7-6700 CPU under Ubuntu 16.04 in an Oracle 64-bit JVM. Xilinx hardware server was of version 2016.1. To confirm platform independence, the framework has also been tested under Windows 8.1. The whole instance occupies 393 MB of memory. Most of this memory usage (307 MB) is caused by the bit mapping required for extracting memory bits from the configuration bitstream. Each of the 6.3 million bits must be mapped individually.

Table 4.30 shows some performance figures of the debugging framework. Parsing the logic location file to create the bit mapping requires a significant amount of time. However, this mapping can be stored in a software cache while the debugger is running. Therefore, this time is only required once before starting the debugging process. Reading out all memory elements of the AMIDAR core requires 664 ms, which is acceptable for interactive debugging. If only the elements of interest are selected, this amount of time reduces to an almost imperceptible value. Retrieving software state information like local variables or fields does not pose any problems for interactive debugging either. The amount of time for executing a line step is mainly limited by the interval for polling the processor state.

Debug Action	Runtime
Parsing the logic location file	1854 ms
Reading memory elements (2x16 bit registers)	12 ms
Reading memory elements (whole AMIDAR core)	664 ms
Reading call stack of a Java thread (5 frames)	2 ms
Reading local variables (1 variable)	2 ms
Reading local variables (9 variables)	4 ms
Reading fields from the heap (1 field)	2 ms
Reading fields from the heap (8 fields)	3 ms
Step execution	200 ms

Table 4.30: Runtime of the debugging framework for certain actions

These figures can be compared to a Vivado ILA core as hardware debugging tool. Such a core has been inserted into the design to monitor the executed code position, stack pointers as well as the interfaces between the AMIDAR data bus and the FUs. This corresponds to 1750 monitored bits, which are also accessible by readback without modifying the design. The ILA core is configured with a minimal sample depth of 1024. After implementation the core occupies about 9.6k look-up tables, 17k registers and 53 BRAMs. Manually triggering the sample process and receiving the samples requires 458 ms. This shows that the readback approach requires much less hardware resources while providing access to almost all memory elements in a similar access time. Even the contents of BRAMs can be obtained, which is not directly possible with an ILA core. On the other hand, the ILA core provides whole signal traces captured in real-time, which cannot be achieved by readback mechanisms.

5 Evaluation

This chapter presents the evaluation results for the AMIDAR processor. Section 5.1 provides a brief overview on the used benchmarks. According to the execution time of each of these benchmarks, the AMIDAR processor is compared with an ARM processor in Section 5.2. Then, the object cache, the garbage collector and the thread scheduler are evaluated in the following three sections respectively. At last, the resource usage of the AMIDAR processor is presented in Section 5.6.

5.1 Benchmarks

Table 5.1 summarizes the benchmarks used for the purpose of the evaluation. SPEC JVM98 is a benchmark suite for measurement of JVM performance, which includes seven programs spanning a range of application characteristics. JOrbis [54] accepts Ogg Vorbis bitstreams and decodes them to raw PCM. VP8Dec [51] converts a VP8 video bitstream to a sequence of reconstructed YUV frames corresponding to the input sequence. SLAM [97] is a program based on the *random sample consensus* (RANSAC) algorithm [34], which builds a map, while at the same time localizing a robot within that map. MNIST [71] trains a simple neural network and then uses it to recognize digits in different images. In the training phase, multiple threads are employed to perform a number of matrix multiplications.

Program	Description
SPEC JVM98:	
jack	Java parser generator.
mpegaudio	MP3 audio decoder.
javac	Java compiler from JDK 1.0.2.
db	Performs multiple database functions on memory resident database.
jess	Java expert shell system based on NASA's CLIPS expert shell system.
compress	Modified Lempel–Ziv (LZW) method.
mtrt	Raytracer with two threads each rendering a scene.
JOrbis	Ogg Vorbis decoder.
VP8Dec	VP8 video decoder.
SLAM	Program realizing simultaneous localization and mapping in robotic navigation.
MNIST	Digit recognizer based on a simple neural network.

Table 5.1: Evaluation benchmarks

5.2 Performance

Processor performance can be evaluated in several different ways. The most common metric is the time required for executing a given program. To measure the execution time of each of the benchmarks above, the system shown in Figure 4.1 at the beginning of Section 4.1 was adopted. The AMIDAR core included in it is configured as illustrated in Table 5.2. The whole system is implemented on an Artix-7 FPGA [75] from Xilinx and operates at 100 MHz Through the SPI interface, an SD card is connected to the system, which is employed to save the input and output files as well as the reference data for the benchmarks.

All benchmarks were also carried out on a *Raspberry Pi* (RBP) computer [86] based on a 700 MHz single-core ARM processor [10]. The operating system installed on the RBP computer is *Raspbian 4.14*

Java stack depth	Heap size	Thread slots	Fat lock slots
4096 words	450 MB	16	64

Table 5.2: Standard AMIDAR system configuration

that includes JRE 8 from Oracle by default. Exploiting a built-in JIT compiler, the JVM contained in the JRE improves the performance of Java programs by compiling bytecodes into native machine code at run time. Due to this, every benchmark was run twice on the RBP computer. In the first run, the benchmark was carried out a total of 9 times repeatedly to ensure that it had been optimized by the JIT compiler thoroughly, while 10 times in the second run. The difference between the durations of these two runs is considered as the best execution time of the benchmark, which can be reached on the RBP computer. Also, with regard to the impact of garbage collection on program performance, the maximum heap size of the JVM is limited to 450 MB.

Table 5.3 summarizes all measurement results. For every benchmark, the former two columns provide the absolute execution times measured on the AMIDAR processor and the RBP computer respectively. The last column corresponds to the ratio between these two values, which is, however, normalized due to the different clock frequencies used by both systems as follows:

$$ratio_{normalized} = \frac{time_{AMIDAR}}{time_{RBP}} * \frac{freq_{AMIDAR}}{freq_{RBP}} \tag{12}$$

Program	AMIDAR	RBP	$\text{Ratio}_{\text{normalized}}$
jack	363658 ms	23527 ms	2.21
mpegaudio	1349349 ms	35251 ms	5.46
javac	3669449 ms	37786 ms	13.87
db	534035 ms	84539 ms	0.90
jess	2975088 ms	25945 ms	16.38
compress	1308129 ms	74156 ms	2.52
mtrt	393600 ms	18619 ms	3.02
JOrbis	2614 ms	1080 ms	0.35
VP8Dec	3129 ms	515 ms	0.87
SLAM	4090 ms	493 ms	1.18
MNIST	1252771 ms	122592 ms	1.46

Table 5.3: Execution time and comparison

The average of the execution time ratios shown in Table 5.3 is 4.38, which indicates that the ARM processor would still be over 4 times faster than the AMIDAR processor even if their clock speeds were the same. However, this average sinks to 1.36 if the four largest ratios that are obtained from the measurements of jess, javac, mpegaudio and mtrt are excluded from the average calculation. According to the workload analysis, there are several key factors that significantly affect the performance of the AMIDAR processor when running these programs, including:

- Invocation of the **`arraycopy`**-method.

- Performing floating-point operations.
- Using large **switch**-statements.

Just like most native methods, the **arraycopy**-method has been rewritten in Java. As a result, the execution times of jess, mpegaudio and javac are increased dramatically because these programs need to invoke the **arraycopy**-method frequently. Table 5.4 shows the percentage of the execution time that each of these programs solely uses for copying arrays when it runs on either of the two hardware systems. As can be seen in the table, even over 90% of the execution time of jess is consumed by invoking the **arraycopy**-method only.

System	jess	mpegaudio	javac
AMIDAR	92.7%	33%	14.8%
RBP	31%	10.8%	5%

Table 5.4: Percentages of execution time used for copying arrays

Also, the ARM processor has a built-in floating-point coprocessor [10] that has been highly optimized. Regarding the measurement results of SciMark 2.0 [95][20], the ARM processor could perform floating-point operations on average 3 times faster than the AMIDAR processor if both of the processors had the same clock speed. Therefore, programs based on floating-point arithmetic like mpegaudio and mtrt can be executed much more efficiently on the ARM processor.

During the recursive construction of an *abstract syntax tree* (AST), the parser of javac executes **switch**-statements intensively to determine the concrete operation that needs to be performed on each AST node according to the node's type. Some of these **switch**-statements include a large number of cases (up to 163). At the bytecode level, a **switch**-statement is translated either to a **lookupswitch** or to a **tableswitch**. Exploiting the binary search algorithm, a JVM can execute the former bytecode in logarithmic time in worst case by making $O(\log n)$ comparisons, where n is the number of cases[21]. In contrast, the latter bytecode can always be executed in constant time independent of the value of n. As mentioned in Section 4.1.5, both of these bytecodes are replaced with **if_icmpeq**-chains upon generating an AXT file. The worst case execution time of an **if_icmpeq**-chain is linearly proportional to n due to the need for $O(n)$ comparisons. To evaluate the performance overhead caused by the inserted **if_icmpeq**-chains, the class files of javac were patched in the same way as an AXT file. Then, the original and patched versions of javac were run on the RPB computer respectively. During these two runs, the JIT compiler was disabled to ensure that neither the original **lookupswitch**- and **tableswitch**-bytecodes nor the inserted **if_icmpeq**-chains were eliminated by the runtime optimization. The measurement result shows that the execution time of javac is increased by 27.5% because of the replacements of **lookupswitch** and **tableswitch**. This indicates that the execution time of javac when run on the AMIDAR processor could be greatly reduced by simply reimplementing these two **switch**-bytecodes using the binary search algorithm rather than the **if_icmpeq** chain.

[20] SciMark2.0 is a Java benchmark for scientific and numerical computing. SciMark 2.0 measures several computational kernels and reports a composite score in approximate MFLOPS.

[21] Some JIT compilers can generate a jump table for a frequently executed **lookupswitch**. This jump table may contain multiple unused entries in between those holding the valid branch offsets. With the help of this jump table, the **lookupswitch** can also be executed in constant time.

5.3 Object Cache

In Section 4.5.3, two cache index generation schemes, namely DMS and XOR6, are discussed and compared based on simulation results. Both of them have been implemented in hardware for the purpose of further research. Through a Verilog parameter, either of these schemes can be selected and built into the object cache on an application-by-application basis. Additionally, the object cache also includes two profiling counters that are employed to count the numbers of cache hits and misses respectively. With the help of these counters, the cache miss rate can be measured precisely at runtime. Table 5.5 summarizes the miss rates of the DMS- and XOR6-based object caches in permille, which were obtained by running the benchmarks introduced in Section 5.1.

Program	DMS	XOR6	Δ% (XOR6 ⟷ DMS)
jack	27.07	26.84	-0.86
mpegaudio	7.07	8.17	13.46
javac	55.67	37.28	-49.33
db	169.91	155.42	-9.32
jess	16.10	15.83	-1.71
compress	65.16	76.39	14.70
mtrt	105.33	136.10	22.61
JOrbis	21.03	28.03	24.97
VP8Dec	44.26	26.09	-69.64
SLAM	40.43	26.75	-51.14
MNIST	38.89	69.76	44.25
Average	53.72	55.15	2.59

Table 5.5: Miss rate comparison

Note that all cache miss rates shown in the table above were measured according to both read and write accesses. Therefore, they differ from the simulation results provided in Section 4.5.3, which correspond to read miss rates only. Also, the AMIDAR simulator has an infinite heap and therefore does not need garbage collection, which can further affect the results measured on it. As Table 5.5 illustrates, neither of the two schemes provides a better miss rate for every benchmark. DMS is more efficient when running mpegaudio, compress, mtrt, JOrbis and MNIST, while XOR6 is more suitable for running the remaining benchmarks. Since the average miss rate provided by DMS is slightly better than that provided by XOR6, it is currently used as the default index generation scheme of the object cache.

5.4 Garbage Collector

5.4.1 Functional Verification

To verify the main functionalities of the garbage collector, a suite of unit tests have been developed. This section presents two of them that are used to test the reference tracing process and the handling of soft reference objects respectively. These tests have been run successfully on AMIDAR systems with different heap sizes from 16 MB to 450 MB.

Reference Tracing Test

This test aims to check if the garbage collector can still detect live objects properly when the heap is highly loaded. At the beginning of the test, two binary object trees of depth 10 are created, which share a common subtree, namely *Tree 3*, as illustrated in Figure 5.1. All nodes included in *Tree 3* are colored green to distinguish them from those of *Tree 1* and *Tree 2*.

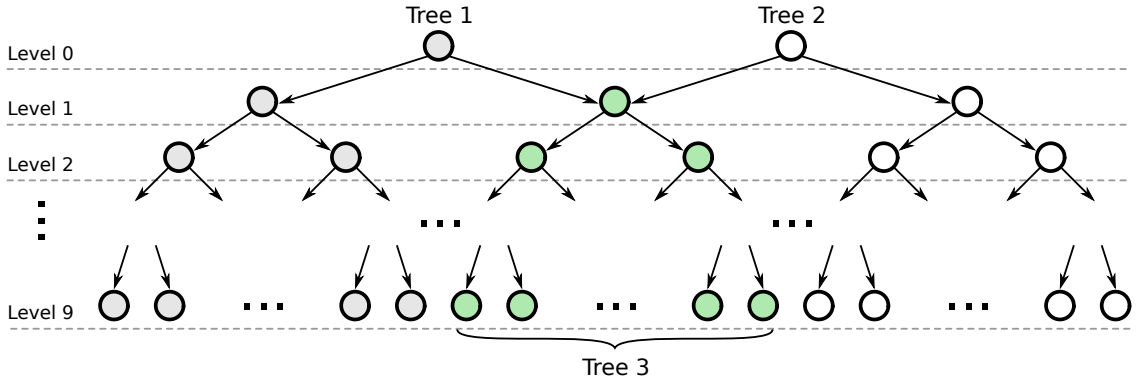


Figure 5.1: Object trees sharing a common subtree

Immediately after the creation of these two trees, the reference to the root of *Tree 1* is deleted, making all gray nodes shown in the figure eligible for garbage collection. Then, a large number of independent object trees with the same depth as *Tree 1* are generated, which totally consume 10 times as much memory space as the size of the heap. This ensures that the garbage collection process will be triggered multiple times. Without receiving any **`OutOfMemoryError`** thrown from the heap manager, the completeness of *Tree 2* is examined at the end of the test.

Soft Reference Test

Soft references are typically utilized to implement memory-sensitive caches which clear cached data automatically when the heap is about to become exhausted. In this test, the life cycles of a set of soft reference objects and their referents are tracked by using such a cache based on an one-dimensional array of type **`SoftReference`**, which has a fixed size (100000 by default). The whole test is performed in the following three steps:

1. The cache is first filled up with soft reference objects that are associated with one common reference queue. The referent of every soft reference object is created from a class called **`Referent`**. This class only contains a static integer field employed to count the number of finalized instances and the overridden **`finalize`**-method that simply increases this field.

2. To trigger a garbage collection cycle, a number of object trees used in the previous test are generated without keeping any references to their roots. In the meantime, the count field of class **`Referent`** is checked periodically. Once it reaches the size of the cache, the test goes to the next step because all referent objects created in the first step have been finalized successfully.

3. As long as the reference queue is empty, the test continues to generate object trees in order to trigger another garbage collection cycle. This is because a soft reference object can only be enqueued after its referent has been finalized. As a result, both operations have to be performed in different

garbage collection cycles. As soon as the reference queue becomes nonempty, the soft reference objects held in it are read out one by one. Each time after getting an object from the queue, the count field of class **`Referent`** is decremented. Once the field becomes zero, the test is complete.

5.4.2 Overhead Analysis

As described in Section 4.5.5, the garbage collector consumes extra processor cycles in both of the mark and compact phases. In the former phase, the entire system has to be blocked, while a cache eviction can be delayed in the latter phase if the object cache needs to access some object being reallocated. To evaluate the impact of the garbage collection process on the processor performance, db from the SPEC JVM98 benchmark suite was analyzed according to several GC-specific metrics. Since running it with the heap size of 450 MB does not trigger any garbage collection cycle, this program was carried out on a customized AMIDAR system with a 64 MB heap in addition. Except the heap size, this system is otherwise the same as the one used for the performance evaluation described above. Table 5.6 lists the relevant measurement results.

GC cycle count	GC trace time	$\text{Runtime}_{64\text{ MB}}$	$\text{Runtime}_{450\text{ MB}}$	GC overhead
2	693 ms	536514 ms	534035 ms	2497 ms

Table 5.6: Measurement results of db

As the table illustrates, two GC cycles are triggered if the heap size is limited to 64 MB. Consequently, the execution time of db is increased by 0.47% (2497 ms). 27.75% (693 ms) of this overhead is caused by tracing references to live objects, while the rest results from compacting the heap. This means that the whole system has to be blocked by the garbage collector completely for a total of 693 ms, namely 0.13% of the entire execution time, when running db. Regarding the count of triggered GC cycles, the average overheads caused by the mark phase, the compact phase and both of them are calculated from the measurement results shown above and summarized in the following table:

$\text{Overhead}_{\text{mark}}$	$\text{Overhead}_{\text{compact}}$	$\text{Overhead}_{\text{total}}$
346.5 ms (0.065%)	902 ms (0.170%)	1248.5 ms (0.235%)

Table 5.7: Average overheads caused by garbage collection

5.5 Thread Scheduler

5.5.1 Functional Verification

In this section, three fundamental unit tests are presented, which are employed to verify the following key functionalities of the thread scheduler: 1. scheduling threads based on the weighted round-robin (WRR) algorithm. 2. synchronizing threads to avoid race conditions. 3. updating thread priorities according to the priority inheritance protocol. On the AMIDAR system used for the performance evaluation, all these tests have been passed successfully.

Thread Scheduling Test

The WRR algorithm schedules multiple threads in a round-robin manner, ensuring the fairness among these threads. Their priorities are reflected by the lengths of time-slices assigned to them. By default, the time-slice of a thread with priority $p + 1$ is twice as long as that of a thread with priority p. In this test, a total of six threads are started in addition to the main thread, among which one has priority 5 (i.e. the default or normal priority), two have priority 6 and the others priority 7. All these threads are created from one common class derived from **Thread**. This class has a static boolean field called **stopped**, which is adopted to terminate all running instances of the class.

Listing 21: Thread task defined in the thread scheduling test

```
1: while(!stopped) {
2:     for(int i = 0; i < 1000; i++)
3:         Double.MAX_VALUE / 1.0 * 1.0;
4:     cnt++;
5:     if(getPriority() == Thread.NORM_PRIORITY && cnt == 100000) {
6:         stopped = true;
7:     }
8: }
```

Listing 21 shows the main part of the overridden **run**-method that simulates the workload of a thread instance at line 2 and 3. After the time-consuming computation, a local variable **cnt** is incremented at line 4, which is intended to keep track of the execution number of the **while**-loop. This number serves as a metric to measure the total processor time assigned to a single thread. After the thread with the normal priority has executed the **while**-loop 10000 times, **stopped** is set to **true**, which terminates the other five threads consequently. The final value of **cnt** of each thread is illustrated in Table 5.8. As can be seen in the table, the processor times are strictly distributed according to the priorities of the six threads.

Thread0 $_{P5}$	Thread1 $_{P6}$	Thread2 $_{P6}$	Thread3 $_{P7}$	Thread4 $_{P7}$	Thread5 $_{P7}$
10000	20022	20023	40030	40032	40034

Table 5.8: Processor times assigned to threads with different priorities

Race Condition Test

This test is based on a simple serial number generator class that solely consists of a static field of type **long** and a **getNext**-method, as shown in Listing 22. Upon calling the **getNext**-method on an instance created from the class, a unique serial number is generated by incrementing **serialNumber**. However, post incrementation in Java is not an atomic operation. Therefore, the **getNext**-method must be thread-safe to avoid race conditions caused by multiple calling threads. This goal is achieved by using a critical section inside the **getNext**-method, which guarantees that only one thread can access **serialNumber** at a time. If any thread preempts the current thread that is executing the **getNext**-method and attempts to

Listing 22: Serial number generator defined in the race condition test

```
1: public class SerialNumberGenerator {
2:     private static long serialNumber = 0;
3:     public long getNext(){
4:         synchronized (this) {
5:             return serialNumber++;
6:         }
7:     }
8: }
```

call this method as well, it will block because it cannot enter the monitor of the generator object. Figure 5.2 demonstrates this situation in detail.

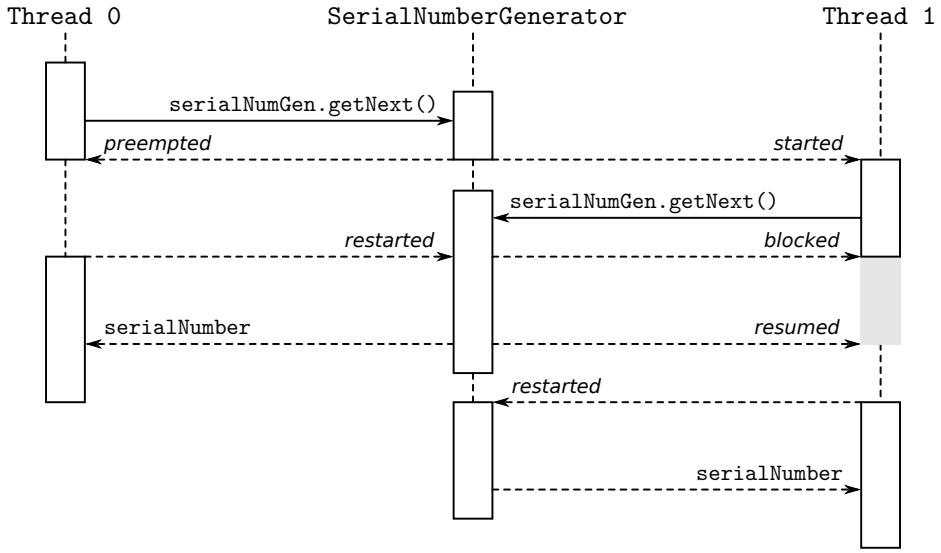


Figure 5.2: Synchronizing two threads

At the beginning of the race condition test, ten racing threads are started, which share a serial number generator and a thread-safe data set. Every thread will generate a total of one million serial numbers throughout its lifetime. Each time after a thread has got a new serial number, it first checks whether the number already exits in the data set. If this is the case, an exception is thrown and the test fails. Otherwise, the number is added into the data set. Only if all racing threads terminate without throwing any exception, the test is considered passed.

Priority Inheritance Test

This test simulates a situation in which priority inheritance occurs as described in the following. First, three threads with priority 5, 6 and 7 are created and started by the main thread successively, which are referred to as $blocker$, $blockee_{P6}$ and $blockee_{P7}$ respectively below. Each time after the main thread has invoked the **`start`**-method on one of the three thread instances, it suspends itself for a while by calling the **`sleep`**-method to allow the newly started thread to run on the processor immediately.

The task performed by $blocker$ is illustrated in Listing 23. As can be seen, once $blocker$ is assigned a time-slice, it first enters two monitors, $m1$ and $m2$, in sequence. Then, it executes a **`while`**-loop as long

Listing 23: Blocker thread task defined in the priority inheritance test

```
 1: synchronized (m1) {
 2:     synchronized (m2) {
 3:         // check point 1
 4:         while(getPriority() != 6) {yield();}
 5:         // check point 2
 6:         while(getPriority() != 7) {yield();}
 7:     }
 8: }
 9: // check point 3
10: if(getPriority() != 5) {throw new RuntimeException();}
```

as it does not inherit the priority of $blockee_{P6}$. This loop is called check point 1. To avoid busy-waiting, *blocker* calls the **yield**-method in each loop iteration. After its priority has been increased to 6, *blocker* proceeds to check point 2 and waits there until it has inherited the priority of $blockee_{P7}$. Subsequently, it releases $m2$ and $m1$ and checks at check point 3 whether its priority has fallen back to the original value, namely 5. If the check fails, an exception will be thrown.

Both of the blockee threads are created from one common class that contains a single field of type **Object**, which is called *m*. This field is initialized in the constructor method of the class, using the only parameter passed in. $blockee_{P6}$ and $blockee_{P7}$ are constructed with $m1$ and $m2$ respectively. The **run**-method of the class solely consists of an empty synchronized block as follows:

Listing 24: Blockee thread task defined in the priority inheritance test

```
1: synchronized (m) {}
```

This implies that $blockee_{P6}$ and $blockee_{P7}$ will block as soon as they start running because the monitors they need have been owned by *blocker*. However, immediately after $blockee_{P7}$ has blocked, *blocker* will leave check point 2 and release both monitors, which makes the two blocked threads become ready again. Figure 5.3 demonstrates the priority changes of *blocker* throughout its entire lifetime. Note that the main thread is not shown in the figure.

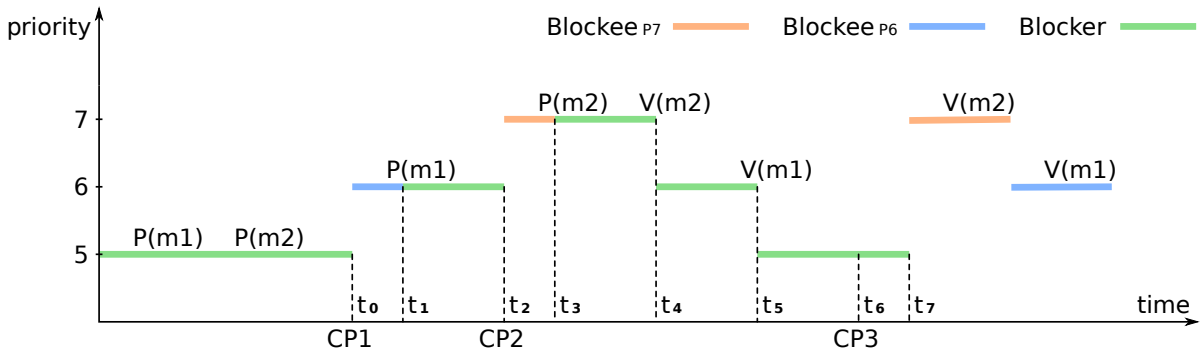


Figure 5.3: Priority changes of the blocker thread

As the figure illustrates, *blocker* reaches check point 1 (t_0) after it has acquired $m1$ and $m2$. At t_1, its priority is changed to 6 as $blockee_{P6}$ fails to enter $m1$. Then, *blocker* gives up the processor at

check point 2 (t_2) so that $blockee_{p7}$ can start running. However, $blockee_{p7}$ blocks at t_3 since it cannot acquire $m2$. As a result, $blocker$ inherits priority 7. Following that, $blocker$ releases $m2$ and $m1$ in sequence, making its priority fall back to 5 at t_5. After it has passed the check at check point 3 (t_6), $blocker$ terminates at t_7.

5.5.2 Overhead Analysis

The key advantage provided by the thread scheduler is that the scheduling process runs in a truly concurrent fashion. Consequently, upon the occurrence of a system tick or an IRQ, context can be switched immediately. This means that the scheduler consumes extra processor cycles only for the purpose of context switching. To measure this overhead, two profiling counters were built into the scheduler. One of them counts the total number of context switches and the other the number of processor cycles used for performing these context switches. Table 5.9 summarizes the measurement results and the statistics calculated based on them.

Program	Context switches	Total CS Cycles	Cycles per CS	Total overhead
jack	276	31619	114.56	8.7E-5%
mpegaudio	77	6767	87.88	5.0E-6%
javac	68	7204	105.94	2.0E-6%
db	556	57058	102.62	1.1E-4%
jess	882	123669	140.21	4.2E-5%
compress	348	43750	125.72	3.3E-5%
mtrt	1531649	282772390	184.62	0.72%
JOrbis	1152	106706	92.63	4.0E-2%
VP8Dec	54	4546	84.19	1.5E-3%
SLAM	47	4180	88.94	1.0E-3%
MNIST	3024798	203684883	67.34	0.16%

Table 5.9: Overheads caused by context switching

All programs shown above, except mtrt and MNIST, are intended to be single-threaded. However, the AMIDAR processor utilizes threads to handle interrupts. This has the consequence that a single-threaded program becomes multi-threaded automatically, if it uses any interrupt-capable peripheral like UART. Currently, UART is employed to realize the standard I/O streams (i.e. **`System.in`** and **`System.out`**). Therefore, the programs listed in Table 5.9 must run the IST of UART in addition because all of them need to output information via the **`print`**- or **`println`**-method.

According to the measurement results, running an originally single-threaded program requires much less context switches than running a multi-threaded one. As a result, the overhead incurred by context switching is negligible for such a program. Due to this, the following discussion solely focuses on mtrt and MNIST. Throughout the entire lifetime of mtrt, only two user threads are started. In contrast, MNIST employs a total of 1100 threads to perform 110 matrix multiplications (i.e. 10 threads per matrix multiplication) in the training phase. Since these matrix multiplications need to be performed sequentially because of data dependencies, up to 10 user threads can run in parallel. As Table 5.9

shows, context switching consumes only 0.72% and 0.16% of the execution times of mtrt and MNIST respectively.

The time required for performing a single context switch is determined by the number and types of the remaining tokens of each FU at the beginning of the context switch, which heavily depends on the characteristic of the running program. Regarding the measurement results above, it takes on average 109 cycles per context switch.

5.6 Resource Usage

Table 5.10 shows the resource usage of the AMIDAR core contained in the evaluation system. All values in the table were measured by using Xilinx Vivado v2017.2 on the Artix-7 FPGA chosen as the standard platform for the AMIDAR processor. As can be seen, 44.03% of the LUTs, 11.59% of the registers and 38.36% of the BRAMs in the FPGA are utilized by the AMIDAR core. This high resource usage results from the hardware implementation of all bytecodes (except both switch bytecodes) and system services. It can be considered as the cost which is paid in exchange for the other benefits of the AMIDAR processor.

LUTs	Registers	BRAMs
59259(44.03%)	31189(11.59%)	140(38.36%)

Table 5.10: Resource usage of AMIDAR core

Table 5.11 illustrates the resource distribution among the FUs inside the AMIDAR core. Clearly, the token machine and the heap manager consume the majority of the LUTs and registers of the core, while the frame stack requires the most BRAMs. Also, the IALU and the FPU consume a significant amount of resources due to the support for 64-bit operations.

FU	LUTs	Registers	BRAMs
Token machine	14300(24.13%)	10095(32.37%)	28(20.00%)
Frame stack	1846(3.12%)	684(2.19%)	69.5(49.64%)
Heap manager	13774(23.24%)	7890(25.30%)	32(22.86%)
Thread scheduler	4179(7.05%)	2429(7.79%)	9(6.43%)
Debugger	794(1.34%)	1359(4.36%)	0
IALU	7887(13.31%)	2203(7.06%)	0
FPU	9419(15.89%)	4081(13.08%)	0

Table 5.11: Resource distribution among FUs inside AMIDAR core

6 Conclusion

This chapter presents a brief overview of this thesis with regard to the research goals defined in Section 1.2. Following that, several suggestions are provided for future research, which should improve the performance and usability of the AMIDAR processor further.

6.1 Summary

This thesis has described the implementation of an AMIDAR-based Java processor whose key characteristics are summarized as follows:

- The AMIDAR processor is the first and currently the only Java processor that is capable of executing a standard JVM benchmark suite (SPEC JVM98) properly[22]. This ensures that a broad range of desktop or even server class applications can run on it, providing a powerful research platform for the AMIDAR project. Obviously, this ability also indicates that the AMIDAR processor has realized all essential functionalities required by a Java runtime system, from bytecode execution to automatic memory and thread management.
- The AMIDAR processor has a compact executable format, namely AXT. Using a single common constant pool, this format eliminates the vast majority of redundant information that is present in the original class files of an application. As a result, the size of the code memory can be greatly reduced, which enables a complex adaption program to be loaded together with a large application into the AMIDAR processor. However, since an AXT file needs to be generated at compile time, the AMIDAR processor currently does not support dynamic class loading and linking.
- The AMIDAR processor has reached an acceptable level of performance in comparison with Oracle JRE 8 running on an ARM processor. As discussed in Section 5.2, its performance could be increased significantly through several simple optimizations like hardware support for array copy.
- The AMIDAR processor includes an efficient object cache that has been implemented based on a novel cache index generation scheme. According to the evaluation results, this scheme provides a better average hit rate than the classical XOR-based scheme.
- The major part of the garbage collector of the AMIDAR processor has been implemented in hardware. This makes it much more efficient than a classical software-based garbage collector. As shown in Section 5.4.2, a complete garbage collection cycle increases on average the execution time of an application by only 0.235%. Exploiting the two equally sized semi-spaces of the dynamic heap, this garbage collector allows an application to run concurrently during the compact phase. Also, due to the built-in object lock mechanism, a DMA transfer can be safely performed at any time, increasing the data throughput. Furthermore, both object finalization and reference objects are supported with the help of a GC-specific thread. Currently, the main weakness of the garbage collector is that it has to stop the world in the mark phase, which reduces the responsiveness of the AMIDAR processor.

[22] Although picoJava-II should also be able to run this benchmark suite, it has never been actually realized in hardware. The FPGA-based prototype presented in [84] is just a partial implementation of the original design, which cannot execute any representative benchmark.

- Thread scheduling and synchronization have been realized in hardware completely. Thus, the only overhead incurred by thread management is the time consumed by context switches, which corresponds to a very small fraction of total execution time, as demonstrated in Section 5.5.2. Additionally, the AMIDAR processor uses a thread-based interrupt handling mechanism that allows external events to be handled in an interactive way.
- A debugging framework has been developed for the AMIDAR processor, which provides powerful debugging support at both software and hardware levels. With the help of this framework, a number of bugs have been located and fixed during the implementation of the AMIDAR processor.

Concluding, it can be stated that all predefined research goals have been fully achieved. Combining with a CGRA-based accelerator, the AMIDAR processor can already speed up hot spots of arbitrary applications dynamically [122].

6.2 Future Work

This section provides several suggestions for improving the performance and usability of the AMIDAR processor, as listed below:

- *Hardware support for array copy*

  As discussed in Section 5.2, the **`arraycopy`**-method has become one of the main performance bottlenecks of the AMIDAR processor. It is not just frequently invoked by applications but also by other API classes such as **`java.lang.String`**. Therefore, it is worth to implement this method in hardware directly. Exploiting the reallocate operation provided by the garbage collector, the **`arraycopy`**-method could be easily realized at the slight expense of hardware overhead.

- *Reimplementation of the switch bytecodes*

  Both of **`lookupswitch`** and **`tableswitch`** should be reimplemented by using the binary search algorithm instead of the plain **`if_icmpeq`**-chain. This simple optimization should increase the execution performance of large **`switch`**-statements significantly.

- *Implementation of a concurrent tracer for the garbage collector*

  For ease of debugging, the current version of the tracer has to work in a stop-the-world manner, reducing the responsiveness of the AMIDAR processor, which could be critical for real-time applications. To allow the tracer to run concurrently with the processor, a write-barrier should be integrated into the object cache according to the Steele's algorithm described in Section 2.3.3. Once an already marked object is modified during the mark phase, the write barrier needs to pass its handle to the tracer. Then, the tracer must push the handles of all objects referenced by the modified object onto the GC stack again. Note that although this extension would improve the responsiveness of the AMIDAR processor, it would also increase the average GC cycle length.

- *Development of a Multi-core Processor*

  The AMIDAR model allows different FUs to be easily integrated into a processor based on it. For this reason, a multi-core processor could be developed by inserting multiple token machines and

frame stacks into the existing AMIDAR processor. The other FUs like IALU and FPU could be simply shared by all cores. In addition, the garbage collector, the thread scheduler and the debugger should also be extended accordingly.

Among all suggested optimizations, the former three could be conducted quickly. In contrast, the last one would require much more effort, through which, however, the largest performance increase should be achieved. Besides these optimizations, a number of weaknesses in the current implementation of the AMIDAR processor should also be addressed, including the inefficient FPU, the limitation of the maximum number of alive threads, the lack of support for dynamic class loading and linking etc. Nevertheless, this thesis has shown that AMIDAR is a promising processor model for tackling today's and tomorrow's problems in the field of embedded systems and is worth exploring further in future research.

Bibliography

[1] Peter Baer Galvin Abraham Silberschatz and Greg Gagne. *Operating System Concepts*. Hoboken, NJ, USA: Wiley, 2008.

[2] J. Adomat, J. Furunas, L. Lindh, and J. Starner. *Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems*. In: *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*. June 1996, pp. 164–168.

[3] Ole Agesen et al. *An Efficient Meta-lock for Implementing Ubiquitous Synchronization*. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. Denver, Colorado, USA: ACM, 1999, pp. 207–222. ISBN: 1-58113-238-7.

[4] J. Agron et al. *Run-Time Services for Hybrid CPU/FPGA Systems on Chip*. In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. Dec. 2006, pp. 3–12.

[5] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. *Static Analyses for Eliminating Unnecessary Synchronization from Java Programs*. In: *Proceedings of the 6th International Symposium on Static Analysis*. SAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 19–38. ISBN: 3-540-66459-9.

[6] *AMIDAR*. `http://www.amidar.de/`.

[7] D. Andrews et al. *Programming models for hybrid FPGA-cpu computational components: a missing link*. In: *IEEE Micro* 24.4 (July 2004), pp. 42–53. ISSN: 0272-1732.

[8] H. Angepat, G. Eads, C. Craik, and D. Chiou. *NIFD: Non-intrusive FPGA Debugger – Debugging FPGA 'Threads' for Rapid HW/SW Systems Prototyping*. In: *2010 International Conference on Field Programmable Logic and Applications*. Aug. 2010, pp. 356–359.

[9] *ARM1136 Technical Reference Manual*. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0211k/DDI0211K_arm1136_r1p5_trm.pdf`. Feb. 2009.

[10] *ARM1176 Technical Reference Manual*. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf`. Nov. 2009.

[11] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. *Thin Locks: Featherweight Synchronization for Java*. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 258–268. ISBN: 0-89791-987-4.

[12] Henry G. Baker Jr. *List Processing in Real Time on a Serial Computer*. In: *Commun. ACM* 21.4 (Apr. 1978), pp. 280–294. ISSN: 0001-0782.

[13] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. *Myths and Realities: The Performance Impact of Garbage Collection*. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, pp. 25–36. ISBN: 1-58113-873-3.

[14] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. *Profile-based Pretenuring*. In: *ACM Trans. Program. Lang. Syst.* 29.1 (Jan. 2007). ISSN: 0164-0925.

[15] Stephen M. Blackburn et al. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 169–190. ISBN: 1-59593-348-4.

[16] Gedare Bloom, Gabriel Parmer, Bhagirath Narahari, and Rahul Simha. *Shared Hardware Data Structures for Hard Real-time Systems*. In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '12. Tampere, Finland: ACM, 2012, pp. 133–142. ISBN: 978-1-4503-1425-1.

[17] *Boehm Garbage Collector*. `http://www.hboehm.info/gc/gcdescr.html`.

[18] Hans-Juergen Boehm and Mark Weiser. *Garbage Collection in an Uncooperative Environment*. In: *Softw. Pract. Exper.* 18.9 (Sept. 1988), pp. 807–820. ISSN: 0038-0644.

[19] Jeff Bogda and Urs Hölzle. *Removing Unnecessary Synchronization in Java*. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. Denver, Colorado, USA: ACM, 1999, pp. 35–46. ISBN: 1-58113-238-7.

[20] U. Brinkschulte, J. Kreuzinger, M. Pfeffer, and T. Ungerer. *A scheduling technique providing a strict isolation of real-time threads*. In: *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*. 2002, pp. 334–340.

[21] R. Brown. *Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem*. In: *Commun. ACM* 31.10 (Oct. 1988), pp. 1220–1227. ISSN: 0001-0782.

[22] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201729881.

[23] Giorgio C. Buttazzo. *Rate Monotonic vs. EDF: Judgment Day*. In: *Real-Time Syst.* 29.1 (Jan. 2005), pp. 5–26. ISSN: 0922-6443.

[24] J. M. Chang and E. F. Gehringer. *A high performance memory allocator for object-oriented systems*. In: *IEEE Transactions on Computers* 45.3 (Mar. 1996), pp. 357–366. ISSN: 0018-9340.

[25] H. J. Chao. *A novel architecture for queue management in the ATM network*. In: *IEEE Journal on Selected Areas in Communications* 9.7 (Sept. 1991), pp. 1110–1118. ISSN: 0733-8716.

[26] J. H. Chao and N. Uzun. *A VLSI sequencer chip for ATM traffic shaper and queue manager*. In: *Global Telecommunications Conference, 1992. Conference Record., GLOBECOM '92. Communication for Global Users., IEEE*. Dec. 1992, 1276–1281 vol.3.

[27] George E. Collins. *A Method for Overlapping and Erasure of Lists*. In: *Commun. ACM* 3.12 (Dec. 1960), pp. 655–657. ISSN: 0001-0782.

[28] *DaCapo*. `http://www.dacapobench.org/`.

[29] *Dalvik Executable Format*. `https://source.android.com/devices/tech/dalvik/dex-format.html`.

[30] David Dice. *Implementing Fast javaTM Monitors with Relaxed-locks*. In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, 2001, pp. 13–13.

[31] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. *On-the-fly Garbage Collection: An Exercise in Cooperation*. In: *Commun. ACM* 21.11 (Nov. 1978), pp. 966–975. ISSN: 0001-0782.

[32] Stefan Döbrich and Christian Hochberger. *Exploring online synthesis for CGRAs with specialized operator sets*. In: *International Journal of Reconfigurable Computing* 2011 (2011), p. 10.

[33] Robert R. Fenichel and Jerome C. Yochelson. *A LISP Garbage-collector for Virtual-memory Computer Systems*. In: *Commun. ACM* 12.11 (Nov. 1969), pp. 611–612. ISSN: 0001-0782.

[34] Martin A. Fischler and Robert C. Bolles. *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography*. In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782.

[35] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown. *A Multithreaded Soft Processor for SoPC Area Reduction*. In: *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Apr. 2006, pp. 131–142.

[36] Etienne M. Gagnon and Laurie J. Hendren. *SableVM: A Research Framework for the Efficient Execution of Java Bytecode*. In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, 2001, pp. 3–3.

[37] A. Garcia, S. Saez, J. Vila, and A. Crespo. *IEEE recommended practice for powering and grounding electronic equipment. (Color Book Series - Emerald Book)*. In: *Proceedings. XII Symposium on Integrated Circuits and Systems Design (Cat. No.PR00387)*. 1999, pp. 78–81.

[38] Stephan Gatzka and Christian Hochberger. *The AMIDAR Class of Reconfigurable Processors*. In: *The Journal of Supercomputing* 32.2 (May 1, 2005), pp. 163–181. ISSN: 1573-0484.

[39] *GDB*. `https://www.gnu.org/software/gdb/`.

[40] P. Graham, B. Nelson, and B. Hutchings. *Instrumenting Bitstreams for Debugging FPGA Circuits*. In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. Mar. 2001, pp. 41–50.

[41] Flavius Gruian and Zoran Salcic. *Designing a Concurrent Hardware Garbage Collector for Small Embedded Systems*. In: *Proceedings of the 10th Asia-Pacific Conference on Advances in Computer Systems Architecture*. ACSAC'05. Singapore: Springer-Verlag, 2005, pp. 281–294. ISBN: 3-540-29643-3, 978-3-540-29643-0.

[42] Marcus Hirt and Marcus Lagergren. *Oracle JRockit: The Definitive Guide*. Packt Publishing, 2010. ISBN: 1847198066, 9781847198068.

[43] Christian Hochberger, Lukas Johannes Jung, Andreas Engel, and Andreas Koch. *Synthilation: JIT-Compilation of Microinstruction Sequences in AMIDAR Processors*. In: *DASIP*. 2014, pp. 193–198.

[44] *HotSpot: A New Breed of Virtual Machine*. `https://www.javaworld.com/article/2076604/hotspot--a-new-breed-of-virtual-machine.html`.

[45] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. *WCET Driven Design Space Exploration of an Object Cache*. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '10. Prague, Czech Republic: ACM, 2010, pp. 26–35. ISBN: 978-1-4503-0122-0.

[46] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. *Worst-case Execution Time Analysis-driven Object Cache Design*. In: *Concurr. Comput. : Pract. Exper.* 24.8 (June 2012), pp. 753–771. ISSN: 1532-0626.

[47] Yousef Iskander, Cameron Patterson, and Stephen Craven. *High-Level Abstractions and Modular Debugging for FPGA Design Validation*. In: *ACM Trans. Reconfigurable Technol. Syst.* 7.1 (Feb. 2014), 2:1–2:22. ISSN: 1936-7406.

[48] *Java Debug Interface Documentation*. `https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/`.

[49] *Java SE 8 Documentation*. `https://docs.oracle.com/javase/8/`.

[50] *Java Virtual Machine Specification*. `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

[51] *JCodec*. `http://jcodec.org/`.

[52] *Jigsaw - W3C's Server*. `https://www.w3.org/Jigsaw/`.

[53] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011. ISBN: 1420082795, 9781420082791.

[54] *JOrbis*. `http://www.jcraft.com/jorbis/`.

[55] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. *Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations*. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '02. Seattle, Washington, USA: ACM, 2002, pp. 130–141. ISBN: 1-58113-471-1.

[56] A. Khan, R. N. Pittman, and A. Forin. *gNOSIS: A Board-Level Debugging and Verification Tool*. In: *2010 International Conference on Reconfigurable Computing and FPGAs*. Dec. 2010, pp. 43–48.

[57] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.

[58] Dirk Koch, Christian Haubelt, and Jürgen Teich. *Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation*. In: *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. FPGA '07. Monterey, California, USA: ACM, 2007, pp. 188–196. ISBN: 978-1-59593-600-4.

[59] P. Kohout, B. Ganesh, and B. Jacob. *Hardware support for real-time operating systems*. In: *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*. Oct. 2003, pp. 45–51.

[60] Thomas Kotzmann and Hanspeter Mössenböck. *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. VEE '05. Chicago, IL, USA: ACM, 2005, pp. 111–120. ISBN: 1-59593-047-7.

[61] Andreas Krall and Mark Probst. *Monitors and exceptions: how to implement Java efficiently*. In: *Concurrency: Practice and Experience* 10.11-13 (1998), pp. 837–850. ISSN: 1096-9128.

[62] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. *A Configurable Hardware Scheduler for Real-Time Systems*. In: *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, 2003, pp. 96–101.

[63] Butler W. Lampson and David D. Redell. *Experience with Processes and Monitors in Mesa*. In: *Commun. ACM* 23.2 (Feb. 1980), pp. 105–117. ISSN: 0001-0782.

[64] P. Lavoie, D. Haccoun, and Y. Savaria. *A systolic architecture for fast stack sequential decoders*. In: *IEEE Transactions on Communications* 42.234 (Feb. 1994), pp. 324–335. ISSN: 0090-6778.

[65] Charles E. Leiserson. *Systolic Priority Queues*. In: *Proceedings of the Caltech Conference On Very Large Scale Integration*. Jan. 1979, pp. 199–214.

[66] Changgong Li, Alexander Schwarz, and Christian Hochberger. *A readback based general debugging framework for soft-core processors*. In: *34th IEEE International Conference on Computer Design, ICCD 2016, Scottsdale, AZ, USA, October 2-5, 2016*. 2016, pp. 568–575.

[67] C. L. Liu and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411.

[68] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall, 2000.

[69] John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782.

[70] *MicroBlaze DM*. `http://www.xilinx.com/support/documentation/ip_documentation/mdm/v3_2/pg115-mdm.pdf`. Nov. 2015.

[71] *MNIST For Machine Learning Beginners*. https://www.tensorflow.org/versions/r1.0/get_started/mnist/beginners.

[72] *ModelSim*. https://www.mentor.com/products/fv/modelsim/.

[73] Takuya Nakaike and Maged M. Michael. *Lock Elision for Read-only Critical Sections in Java*. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 269–278. ISBN: 978-1-4503-0019-3.

[74] Scott Nettles, James O'Toole, and David Pierce. *Replication-Based Incremental Copying Collection*. In: *Proceedings of the International Workshop on Memory Management*. IWMM '92. London, UK, UK: Springer-Verlag, 1992, pp. 357–364. ISBN: 3-540-55940-X.

[75] *Nexys Video Artix-7 FPGA Board*. https://store.digilentinc.com/nexys-video-artix-7-fpga-trainer-board-for-multimedia-applications/.

[76] Kelvin D. Nilsen and William J. Schmidt. *Cost-effective Object Space Management for Hardware-assisted Real-time Garbage Collection*. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 338–354. ISSN: 1057-4514.

[77] J. M. O'Connor and M. Tremblay. *picoJava-I: the Java virtual machine in hardware*. In: *IEEE Micro* 17.2 (Mar. 1997), pp. 45–53. ISSN: 0272-1732.

[78] Tamiya Onodera and Kiyokuni Kawachiya. *A Study of Locking Objects with Bimodal Fields*. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. Denver, Colorado, USA: ACM, 1999, pp. 223–237. ISBN: 1-58113-238-7.

[79] D. Picker and R. D. Fellman. *A VLSI priority packet queue with inheritance and overwrite*. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3.2 (June 1995), pp. 245–253. ISSN: 1063-8210.

[80] *picoJava-II Microarchitecture Guide*. Sun Microsystems. Mar. 1999.

[81] *picoJava-II Programmer's Reference Manual*. Sun Microsystems. Mar. 1999.

[82] Thomas B. Preußer, Peter Reichel, and Rainer G. Spallek. *An Embedded GC Module with Support for Multiple Mutators and Weak References*. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems*. ARCS'10. Hannover, Germany: Springer-Verlag, 2010, pp. 25–36. ISBN: 3-642-11949-2, 978-3-642-11949-1.

[83] Thomas B. Preußer, Martin Zabel, and Peter Reichel. *The SHAP microarchitecture and Java virtual machine*. Tech. rep. TUD-FI07-02. Fakultät Informatik, Technische Universität Dresden, Apr. 2007.

[84] Wolfgang Puffitsch and Martin Schoeberl. *picoJava-II in an FPGA*. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '07. Vienna, Austria: ACM, 2007, pp. 213–221. ISBN: 978-1-59593-813-8.

[85] Ravi Rajwar and James R. Goodman. *Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution*. In: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 34. Austin, Texas: IEEE Computer Society, 2001, pp. 294–305. ISBN: 0-7695-1369-7.

[86] *Raspberry Pi*. https://www.raspberrypi.org.

[87] Jennifer Rexford, John Hall, and Kang Shin. *A Router Architecture for Real-time Point-to-point Networks*. In: *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. ISCA '96. Philadelphia, Pennsylvania, USA: ACM, May 1996, pp. 237–246. ISBN: 0-89791-786-3.

[88] Erik Ruf. *Effective Synchronization Removal for Java*. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 208–218. ISBN: 1-58113-199-2.

[89] S. Saez, J. Vila, A. Crespo, and A. Garcia. *A hardware scheduler for complex real-time systems*. In: *Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on*. Vol. 1. 1999, 43–48 vol.1.

[90] *Sauter AG*. https://www.sauter-controls.com/en.html.

[91] William J. Schmidt and Kelvin D. Nilsen. *Performance of a Hardware-assisted Real-time Garbage Collector*. In: *SIGOPS Oper. Syst. Rev.* 28.5 (Nov. 1994), pp. 76–85. ISSN: 0163-5980.

[92] Martin Schoeberl. *A Java Processor Architecture for Embedded Real-time Systems*. In: *J. Syst. Archit.* 54.1-2 (Jan. 2008), pp. 265–286. ISSN: 1383-7621.

[93] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. VDM - Verlag Dr. Müller, 2008. ISBN: 978-3-8364-8086-4.

[94] Martin Schoeberl. *JOP Reference Handbook*. CreateSpace, 2009. ISBN: 978-1-4382-3969-9.

[95] *SciMark 2.0*. https://math.nist.gov/scimark2.

[96] L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority inheritance protocols: an approach to real-time synchronization*. In: *IEEE Transactions on Computers* 39.9 (Sept. 1990), pp. 1175–1185. ISSN: 0018-9340.

[97] *SLAM*. https://github.com/cberzan/bbr-mapper/tree/master/slam.

[98] *SPEC jbb2005*. https://www.spec.org/jbb2005/.

[99] *SPEC JVM98*. https://www.spec.org/jvm98/.

[100] W. Srisa-an, C. T. D. Lo, and J. M. Chang. *Active memory processor: a hardware garbage collector for real-time Java embedded devices*. In: *IEEE Transactions on Mobile Computing* 2.2 (Apr. 2003), pp. 89–101. ISSN: 1536-1233.

[101] William Stallings. *Operating Systems: Internals and Design Principles*. Vol. 8. Pearson, 2015.

[102] Guy L. Steele Jr. *Multiprocessing Compactifying Garbage Collection*. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 495–508. ISSN: 0001-0782.

[103] Neal Stollon. *On-Chip Instrumentation: Design and Debug for Systems on Chip*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1441975624, 9781441975621.

[104] Tórur Biskopstø Strøm, Wolfgang Puffitsch, and Martin Schoeberl. *Chip-multiprocessor Hardware Locks for Safety-critical Java*. In: *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '13. Karlsruhe, Germany: ACM, 2013, pp. 38–46. ISBN: 978-1-4503-2166-2.

[105] Y. Tang and N. W. Bergmann. *A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems*. In: *IEEE Transactions on Computers* 64.5 (May 2015), pp. 1254–1267. ISSN: 0018-9340.

[106] *Target Communication Framework*. http://www.eclipse.org/tcf/.

[107] *The CACAO Virtual Machine*. http://www.cacaojvm.org/.

[108] *The Kaffe Virtual Machine*. http://www.kaffe.org/.

[109] *The Sable Virtual Machine*. http://www.sablevm.org/.

[110] Anurag Tiwari and Karen A. Tomko. *Scan-chain Based Watch-points for Efficient Run-time Debugging and Verification of FPGA Designs*. In: *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. ASP-DAC '03. Kitakyushu, Japan: ACM, 2003, pp. 705–711. ISBN: 0-7803-7660-9.

[111] K. Toda, K. Nishida, E. Takahashi, and Y. Yamaguchi. *A priority forwarding router chip for real-time interconnection networks*. In: *1994 Proceedings Real-Time Systems Symposium*. Dec. 1994, pp. 63–73.

[112] D. M. Tullsen, S. J. Eggers, and H. M. Levy. *Simultaneous multithreading: Maximizing on-chip parallelism*. In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. June 1995, pp. 392–403.

[113] Sascha Uhrig and Jörg Wiese. *Jamuth: An IP Processor Core for Embedded Java Real-time Systems*. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '07. Vienna, Austria: ACM, 2007, pp. 230–237. ISBN: 978-1-59593-813-8.

[114] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 1402065043, 9781402065040.

[115] N. Vijaykrishnan and N. Ranganathan. *Supporting object accesses in a Java processor*. In: *Computers and Digital Techniques, IEE Proceedings* - 147.6 (Nov. 2000), pp. 435–443. ISSN: 1350-2387.

[116] Narayanan Vijaykrishnan, N. Ranganathan, and Ravi Gadekarla. *Object-Oriented Architectural Support for a Java Processor*. In: *Proceedings of the 12th European Conference on Object-Oriented Programming*. ECCOP '98. London, UK, UK: Springer-Verlag, 1998, pp. 330–354. ISBN: 3-540-64737-6.

[117] *Vivado Debug Core*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug908-vivado-programming-debugging.pdf. Mar. 2014.

[118] Timothy Wheeler, Paul Graham, Brent E. Nelson, and Brad Hutchings. *Using Design-Level Scan to Improve FPGA Design Observability and Controllability for Functional Verification*. In: *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*. FPL '01. London, UK, UK: Springer-Verlag, 2001, pp. 483–492. ISBN: 3-540-42499-7.

[119] R. Willenberg and P. Chow. *Simulation-based HW/SW co-debugging for field-programmable systems-on-chip*. In: *2013 International Conference on Field programmable Logic and Applications*. Sept. 2013, pp. 1–8.

[120] Ifor Williams and Mario Wolczko. *An object-based memory architecture*. In: *Proceedings of the Fourth International Workshop on Persistent Object Systems*. Martha's Vineyard, MA (USA), Sept. 1990, pp. 114–130.

[121] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. In: *Proceedings of the International Workshop on Memory Management*. IWMM '92. London, UK, UK: Springer-Verlag, 1992, pp. 1–42. ISBN: 3-540-55940-X.

[122] Dennis L. Wolf, Lukas J. Jung, Tajas Ruschke, Changgong Li, and Christian Hochberger. *AMIDAR Project: Lessons Learned in 15 Years of Researching Adaptive Processors*. In: *13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2018, Lille, France, July 9-11, 2018*. 2018, pp. 1–8.

[123] Greg Wright, Matthew L. Seidl, and Mario Wolczko. *An Object-aware Memory Architecture*. In: *Sci. Comput. Program.* 62.2 (Oct. 2006), pp. 145–163. ISSN: 0167-6423.

[124] *Xilinx 7 Series FPGAs Memory Interface Solutions*. https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_1/ug586_7Series_MIS.pdf. Apr. 2018.

[125] *Xilinx CAM*. https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf. Mar. 2011.

[126] *Xilinx ChipScope Pro*. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/ug750.pdf. Mar. 2013.

[127] Chi Hang Yau, Yi Yu Tan, Anthony S. Fong, and Wing Shing Yu. *Hardware Concurrent Garbage Collection for Short-Lived Objects in Mobile Java Devices*. In: *Embedded and Ubiquitous Computing – EUC 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 47–56. ISBN: 978-3-540-32295-5.

[128] T. Yuasa. *Real-time Garbage Collection on General-purpose Machines*. In: *J. Syst. Softw.* 11.3 (Mar. 1990), pp. 181–198. ISSN: 0164-1212.

[129] Martin Zabel, Thomas B. Preußer, and Rainer G. Spallek. *Increasing the Efficiency of an Embedded Multi-core Bytecode Processor Using an Object Cache*. In: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '12. Copenhagen, Denmark: ACM, 2012, pp. 88–97. ISBN: 978-1-4503-1688-0.

[130] *ZPU*. `https://opencores.org/project/zpu`.

A Additional Measurement Results

To gain insights into various aspects of the AMIDAR processor, the following values were measured on the evaluation system described in Section 5.2 by running the benchmarks introduced in Section 5.1:

- Instruction cache miss rate.
- Maximum Java stack depth.
- Total number of triggered garbage collection cycles.
- Peak number of handles used at the same time.
- Handle table cache miss rate.

Table A.1 summarizes the measurement results. Note that the miss rate of the instruction cache corresponds to the number of cache misses per 10^7 accesses, while that of the handle table cache is given in permille. Additionally, the peak number of handles also denotes the total number of allocated objects, if no garbage collection cycle is triggered.

Program	Miss rate of I-cache	Max. Java stack depth	GC cycle number	Peak handle number	Miss rate of HT-cache
jack	241	712	2	3738345	59.70
mpegaudio	150	274	0	86126	0.77
javac	2182	2433	28	2865494	70.25
db	19	315	0	3308701	270.75
jess	4	976	5	3701813	21.70
compress	7	292	1	80967	0.06
mtrt	79	293	1	3721088	156.81
JOrbis	2372	191	0	21257	7.41
VP8Dec	2048	169	0	60647	37.54
SLAM	2229	179	0	52372	81.61
MNIST	16	169	0	116992	0.05

Table A.1: Measurement results for each benchmark

B FU Operations

B.1 Token Machine Operations

This section provides an overview of the operations supported by the token machine. Both data input ports of the token machine are referred to as $port\_0$ and $port\_1$ respectively below.

ANEWARRAY_INFO
Operand count: 1
Operand 0: 16-bit CTI of an array type held in $port\_0[15:0]$
Result: {Flags, CTI}
Description: The 16 flag bits of the array type are read out from the class table and sent together with the given CTI as a 32-bit result to the target FU.

BRANCH
Operand count: 1
Operand 0: branch offset held in $port\_0[31:0]$
Result: void
Description: An unconditional jump is performed by using the branch offset provided by operand 0.

BRANCH_IF_EQ
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$
Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is equal to zero.

BRANCH_IF_GE
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$
Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is greater than or equal to zero.

BRANCH_IF_GT
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$
Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is greater than zero.

BRANCH_IF_LE
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$
Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is less than or equal to zero.

BRANCH_IF_LT
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$
Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is less than zero.

BRANCH_IF_NE
Operand count: 2
Operand 0: comparison result held in $port\_0[31:0]$
Operand 1: branch offset held in $port\_1[31:0]$

Result: void
Description: A jump is performed by using the branch offset provided by operand 1, if operand 0 is not equal to zero.

CHECKCAST
Operand count: 2
Operand 0: $\text{CTI}_{\text{Class A}}$ held in $port\_0[31:0]$
Operand 1: $\text{CTI}_{\text{Class B}}$ held in $port\_1[31:0]$
Result: void
Description: Nothing happens if both CTIs are equal or $\text{CTI}_{\text{Class A}}$ belongs to a subclass of B; otherwise, a hardware exception is thrown.

FORCESCHEDULING
Operand count: 0
Result: void
Description: The decoding pipeline stops fetching new bytecode immediately and the $CS\_waiting$ signal is asserted to request a context switch.

INSTANCE_OF
Operand count: 2
Operand 0: $\text{CTI}_{\text{Class A}}$ held in $port\_0[31:0]$
Operand 1: $\text{CTI}_{\text{Class B}}$ held in $port\_1[31:0]$
Result: an integer value indicating whether A and B are of the same type
Description: The result is set to one if both CTIs are equal or $\text{CTI}_{\text{Class A}}$ belongs to a subclass of B; otherwise, the result is set to zero.

INVOKE
Operand count: 1
Operand 0: 16-bit CTI of a class held in $port\_0[15:0]$
Result: {Num_arg, Max_locals, AMTI, PC}
Description: An instance method is invoked according to the CTI provided by operand 0 and the RMTI provided by a previously executed `LOAD_ARG_RMTI` operation. A 64-bit result is sent to the target FU (currently, only the frame stack). The high 32 bits of the result consist of the argument number and the maximum number of the local variables of the invoked method. The low 32 bits of the result are made up of the AMTI and the PC of the invoking method.

INVOKE_INTERFACE
Operand count: 1
Operand 0: 16-bit CTI of a class held in $port\_0[15:0]$
Result: {Num_arg, Max_locals, AMTI, PC}
Description: An interface method is invoked according to the CTI provided by operand 0 as well as the IOLI of the corresponding interface and the declaration order of the method provided by a previously executed `LOAD_ARG_IOLI_RIMTI` operation. A 64-bit result is sent to the target FU (currently, only the frame stack). The high 32 bits of the result consist of the argument number and the maximum number of the local variables of the invoked method. The low 32 bits of the result are made up of the AMTI and the PC of the invoking method.

INVOKE_STATIC
Operand count: 1
Operand 0: 16-bit AMTI of a static method held in $port\_0[15:0]$

Result: {Num_arg, Max_locals, AMTI, PC}
Description: A static method is invoked according to the AMTI provided by operand 0. A 64-bit result is sent to the target FU (currently, only the frame stack). The high 32 bits of the result consist of the argument number and the maximum number of the local variables of the invoked method. The low 32 bits of the result are made up of the AMTI and the PC of the invoking method.

JSR
Operand count: 1
Operand 0: branch offset held in $port\_0[31:0]$
Result: the PC of the next bytecode
Description: An unconditional jump to a subroutine is performed by using the branch offset provided by operand 0. At the same time, the PC of the bytecode immediately following the current one is sent to the target FU (currently, only the frame stack), which is used as the return address of the subroutine.

LDC
Operand count: 1
Operand 0: 16-bit constant pool index held in $port\_0[15:0]$
Result: the constant pool entry at the given index
Description: The corresponding constant pool entry is read out and sent to the target FU.

LDC2
Operand count: 1
Operand 0: 16-bit constant pool index held in $port\_0[15:0]$
Result: the two successive constant pool entries at the given index
Description: The corresponding constant pool entries are read out and sent together as a 64-bit result to the target FU.

LOAD_ARG_RMTI
Operand count: 2
Operand 0: 6-bit argument number held in $port\_0[15:10]$
Operand 1: 10-bit RMTI of a virtual method held in $port\_0[9:0]$
Result: void
Description: The operands are loaded into internal registers in order to perform an `INVOKE` operation immediately following the `LOAD_ARG_RMTI` operation.

LOAD_ARG_IOLI_RIMTI
Operand count: 3
Operand 0: 6-bit argument number held in $port\_0[31:26]$
Operand 1: 10-bit IOLI of an interface held in $port\_0[25:16]$
Operand 2: 16-bit declaration order of a method of the interface held in $port\_0[15:0]$
Result: void
Description: The operands are loaded into internal registers in order to perform an `INOVKE_INTERFACE` operation immediately following the `LOAD_ARG_IOLI_RIMTI` operation.

OBJSIZE
Operand count: 1
Operand 0: 16-bit CTI of a class held in $port\_0[15:0]$
Result: the size of an object created from the class
Description: The result is read out from the class table via the given CTI and sent to the target FU.

NEWARRAY_INFO
Operand count: 1
Operand 0: 8-bit IOLAT of an array type held in $port\_0[7:0]$
Result: {Flags, CTI}
Description: The given IOLAT is first converted to the CTI of the array type. Then, the 16 flag bits of the array type are read out from the class table via the generated CTI and sent together with the CTI as a 32-bit result to the target FU.

RET
Operand count: 1
Operand 0: PC held in $port\_0[15:0]$
Result: void
Description: An unconditional jump to the given PC is performed.

RETURN
Operand count: 2
Operand 0: AMTI held in $port\_0[31:16]$
Operand 1: PC held in $port\_0[15:0]$
Result: void
Description: The token machine starts fetching bytecodes from the new position determined by both of the operands.

SENDBYTECODE_1
Operand count: 0
Result: the first byte immediately following the bytecode being executed
Description: The corresponding byte is extracted from the bytecode stream and sent to the target FU.

SENDBYTECODE_1_2
Operand count: 0
Result: the first two bytes immediately following the bytecode being executed
Description: The corresponding bytes are extracted from the bytecode stream and sent to the target FU.

SENDBYTECODE_1_2_3_4
Operand count: 0
Result: the first four bytes immediately following the bytecode being executed
Description: The corresponding bytes are extracted from the bytecode stream and sent to the target FU.

SENDBYTECODE_2
Operand count: 0
Result: the second byte immediately following the bytecode being executed
Description: The corresponding byte is extracted from the bytecode stream and sent to the target FU.

SENDBYTECODE_3
Operand count: 0
Result: the third byte immediately following the bytecode being executed
Description: The corresponding byte is extracted from the bytecode stream and sent to the target FU.

SENDBYTECODE_3_4
Operand count: 0
Result: the third and fourth bytes immediately following the bytecode being executed

Description: The corresponding bytes are extracted from the bytecode stream and sent to the target FU.

THREADSWITCH
Operand count: 0
Result: the next Thread ID
Description: The next thread ID is sent to the target FU (currently, only the frame stack) immediately after a context switch.

THROW
Operand count: 2
Operand 0: handle of an exception object held in $port\_0[31:0]$
Operand 1: CTI of an exception class held in $port\_1[31:0]$
Result: the handle of the exception object
Description: The exception handling process described in Section 4.3.4 is triggered.

B.2 Frame Stack Operations

This section introduces the operations supported by the frame stack briefly. The single data input port of the frame stack is referred to as $port\_0$ below. Also, where it is clear by context, the operand stack of the current frame is referred to as simply the operand stack.

ACONST_NULL
Operand count: 0
Result: void
Description: A null reference (i.e. handle 0) is pushed onto the operand stack.

CLEARFRAME
Operand count: 0
Result: void
Description: The current frame is reset to its initial state, i.e. the entire operand stack is cleared.

DCONST_<0|1>
Operand count: 0
Result: void
Description: The **`double`** constant <0.0|1.0> is pushed onto the operand stack.

DUP
Operand count: 0
Result: void
Description: The 32-bit value on the top of the operand stack is duplicated.

DUP_X1
Operand count: 0
Result: void
Description: A copy of the 32-bit top value is inserted into the operand stack two 32-bit values from the top.

DUP_X2
Operand count: 0
Result: void

Description: A copy of the 32-bit top value is inserted into the operand stack three 32-bit values from the top.

DUP2
Operand count: 0
Result: void
Description: The two 32-bit values on the top of the operand stack are duplicated.

DUP2_X1
Operand count: 0
Result: void
Description: The two 32-bit values on the top of the operand stack are duplicated and inserted three 32-bit values down in the operand stack.

DUP2_X2
Operand count: 0
Result: void
Description: The two 32-bit values on the top of the operand stack are duplicated and inserted four 32-bit values down in the operand stack.

FCONST_<0|1|2>
Operand count: 0
Result: void
Description: The **`float`** constant <0.0|1.0|2.0> is pushed onto the operand stack.

ICONST_M1
Operand count: 0
Result: void
Description: The **`int`** constant -1 is pushed onto the operand stack.

ICONST_<0|1|2|3|4|5>
Operand count: 0
Result: void
Description: The **`int`** constant <0|1|2|3|4|5> is pushed onto the operand stack.

INVOKE
Operand count: 4
Operand 0: 6-bit argument number of the callee held in $port\_0[53:48]$
Operand 1: 16-bit local variable number of the callee held in $port\_0[47:32]$
Operand 2: 16-bit AMTI of the caller held in $port\_0[31:16]$
Operand 3: 16-bit PC of the caller held in $port\_0[15:0]$
Result: void
Description: A new stack frame is created for the callee as described in Section 4.4.2.

LCONST_<0|1>
Operand count: 0
Result: void
Description: The **`long`** constant <0|1> is pushed onto the operand stack.

LOAD32
Operand count: 1
Operand 0: index of a 32-bit local variable held in $port\_0[31:0]$
Result: void
Description: The 32-bit local variable at the given index is pushed onto the operand stack.

LOAD32_<0|1|2|3>
Operand count: 0
Result: void
Description: The 32-bit local variable at index <0|1|2|3> is pushed onto the operand stack.

LOAD64
Operand count: 1
Operand 0: index of a 64-bit local variable held in $port\_0[31:0]$
Result: void
Description: The 64-bit local variable at the given index is pushed onto the operand stack.

LOAD64_<0|1|2|3>
Operand count: 0
Result: void
Description: The 64-bit local variable at index <0|1|2|3> is pushed onto the operand stack.

PEEK
Operand count: 1
Operand 0: index of an operand stack entry held in $port\_0[31:0]$
Result: the 32-bit value at the given index
Description: The 32-bit value at the given index is read out from the operand stack and sent to the target FU. The index must be greater than or equal to 1, where index 1 refers to the top value. Note that the value still stays on the operand stack after this operation.

PEEK_1
Operand count: 0
Result: the 32-bit value on the top of the operand stack
Description: The 32-bit top value is read out from the operand stack and sent to the target FU. Note that the value still stays on the operand stack after this operation.

POP32
Operand count: 0
Result: the 32-bit value on the top of the operand stack
Description: The 32-bit top value is popped from the operand stack and sent to the target FU.

POP64
Operand count: 0
Result: the 64-bit value on the top of the operand stack
Description: The 64-bit top value is popped from the operand stack and sent to the target FU.

PUSH32
Operand count: 1
Operand 0: 32-bit numeric value held in $port\_0[31:0]$
Result: void

Description: The 32-bit numeric value provided by operand 0 is pushed onto the operand stack.

PUSH64
Operand count: 1
Operand 0: 64-bit numeric value held in $port\_0[63:0]$
Result: void
Description: The 64-bit numeric value provided by operand 0 is pushed onto the operand stack.

PUSHREF
Operand count: 1
Operand 0: 32-bit object handle held in $port\_0[31:0]$
Result: void
Description: The object handle provided by operand 0 is pushed onto the operand stack.

REMOVE32
Operand count: 0
Result: void
Description: The 32-bit top value is removed from the operand stack.

REMOVE64
Operand count: 0
Result: void
Description: The 64-bit top value is removed from the operand stack.

RETURN
Operand count: 0
Result: {AMTI, PC}
Description: The current frame is discarded. In addition, the AMTI and PC of the caller are sent to the target FU (currently, only the token machine).

RETURN32
Operand count: 0
Result: {AMTI, PC}
Description: The current frame is discarded and a 32-bit value is pushed onto the operand stack of the caller. In addition, the AMTI and PC of the caller are sent to the target FU (currently, only the token machine).

RETURN64
Operand count: 0
Result: {AMTI, PC}
Description: The current frame is discarded and a 64-bit value is pushed onto the operand stack of the caller. In addition, the AMTI and PC of the caller are sent to the target FU (currently, only the token machine).

STORE32
Operand count: 1
Operand 0: index of a 32-bit local variable held in $port\_0[31:0]$
Result: void
Description: The 32-bit value on the top of the operand stack is stored into the local variable at the given index.

STORE32_<0|1|2|3>
Operand count: 0
Result: void
Description: The 32-bit value on the top of the operand stack is stored into the local variable at index <0|1|2|3>.

STORE64
Operand count: 1
Operand 0: index of a 64-bit local variable held in $port\_0[31:0]$
Result: void
Description: The 64-bit value on the top of the operand stack is stored into the local variable at the given index.

STORE64_<0|1|2|3>
Operand count: 0
Result: void
Description: The 64-bit value on the top of the operand stack is stored into the local variable at index <0|1|2|3>.

SWAP
Operand count: 0
Result: void
Description: The two 32-bit values on the top of the operand stack are swapped.

THREADSWITCH
Operand count: 1
Operand 0: ID of the next thread held in $port\_0[31:0]$
Result: void
Description: A context switch is performed inside the frame stack according to the next thread ID provided by operand 0.

B.3 Heap Manager Operations

This section presents the operation set of the heap manager. The three data input ports of the heap manager are referred to as $port\_0$, $port\_1$ and $port\_2$ respectively below.

ALLOC_OBJ
Operand count: 2
Operand 0: 16-bit CTI held in $port\_0[15:0]$
Operand 1: 16-bit object size held in $port\_1[15:0]$
Result: the handle of the allocated object
Description: A new object is allocated from the dynamic heap and its handle is sent to the target FU (currently, only the frame stack).

ALLOC_ARRAY
Operand count: 3
Operand 0: 16 flag bits held in $port\_0[31:16]$
Operand 1: 16-bit CTI held in $port\_0[15:0]$
Operand 2: 32-bit array length held in $port\_1[31:0]$
Result: the handle of the allocated array

Description: A new one-dimensional array is allocated from the dynamic heap and its handle is sent to the target FU (currently, only the frame stack).

ALLOC_MULTI_ARRAY
Operand count: 0
Result: the handle of the allocated array
Description: A new multi-dimensional array is allocated from the dynamic heap and its handle is sent to the target FU (currently, only the frame stack). The CTI and the number of the dimensions of the array have been loaded into internal registers by a previously executed `SETUP_MULTI_ARRAY` operation. Also, the size of each dimension has been loaded into a dedicated internal register by a previously executed `SET_MULTI_ARRAY_DIM_SIZE` operation.

FLUSH
Operand count: 1
Operand 0: handle of an object held in $port\_0[31:0]$
Result: void
Description: All cache blocks holding the given object are flushed.

GC_LOCK
Operand count: 1
Operand 0: handle of an object held in $port\_0[31:0]$
Result: void
Description: The given object is locked so that the garbage collector may not reallocate it.

GET_CTI
Operand count: 1
Operand 0: handle of an object held in $port\_0[31:0]$
Result: the CTI of the corresponding class
Description: The CTI of the class from which the object has been created is read out from the handle table cache and is sent to the target FU.

GET_SIZE
Operand count: 1
Operand 0: handle of an object held in $port\_0[31:0]$
Result: the size of the object
Description: The size of the object is read out from the handle table cache and is sent to the target FU.

HO_READ
Operand count: 2
Operand 0: handle of an object held in $port\_0[31:0]$
Operand 1: offset of a 32-bit field of the object held in $port\_1[31:0]$
Result: the value of the field
Description: The 32-bit object field is read-accessed through the given handle-offset pair and its value is sent the target FU.

HO_READ_64
Operand count: 2
Operand 0: handle of an object held in $port\_0[31:0]$
Operand 1: offset of a 64-bit field of the object held in $port\_1[31:0]$
Result: the value of the field

Description: The 64-bit object field is read-accessed through the given handle-offset pair and its value is sent the target FU.

HO_READ_ARRAY
Operand count: 2
Operand 0: handle of an array held in $port\_0[31:0]$
Operand 1: offset (i.e. index) of a 32-bit element of the array held in $port\_1[31:0]$
Result: the value of the element
Description: The 32-bit array element is read-accessed through the given handle-offset pair and its value is sent the target FU. If the index is not within the bounds of the array, a hardware exception is thrown.

HO_READ_ARRAY_64
Operand count: 2
Operand 0: handle of an array held in $port\_0[31:0]$
Operand 1: offset (i.e. index) of a 64-bit element of the array held in $port\_1[31:0]$
Result: the value of the element
Description: The 64-bit array element is read-accessed through the given handle-offset pair and its value is sent the target FU. If the index is not within the bounds of the array, a hardware exception is thrown.

HO_WRITE
Operand count: 3
Operand 0: handle of an object held in $port\_0[31:0]$
Operand 1: offset of a 32-bit field of the object held in $port\_1[31:0]$
Operand 2: 32-bit value held in $port\_2[31:0]$
Result: void
Description: The 32-bit value provided by operand 2 is written into the object filed through the given handle-offset pair.

HO_WRITE_64
Operand count: 3
Operand 0: handle of an object held in $port\_0[31:0]$
Operand 1: offset of a 32-bit field of the object held in $port\_1[31:0]$
Operand 2: 64-bit value held in $port\_2[63:0]$
Result: void
Description: The 64-bit value provided by operand 2 is written into the object filed through the given handle-offset pair.

HO_WRITE_ARRAY
Operand count: 3
Operand 0: handle of an array held in $port\_0[31:0]$
Operand 1: offset (i.e. index) of a 32-bit element of the array held in $port\_1[31:0]$
Operand 2: 32-bit value held in $port\_2[31:0]$
Result: void
Description: The 32-bit value provided by operand 2 is written into the array element through the given handle-offset pair. If the index is not within the bounds of the array, a hardware exception is thrown.

HO_WRITE_ARRAY_64
Operand count: 3
Operand 0: handle of an array held in $port\_0[31:0]$
Operand 1: offset (i.e. index) of a 64-bit element of the array held in $port\_1[31:0]$

Operand 2: 64-bit value held in $port\_2[63:0]$
Result: void
Description: The 64-bit value provided by operand 2 is written into the array element through the given handle-offset pair. If the index is not within the bounds of the array, a hardware exception is thrown.

PHY_READ
Operand count: 1
Operand 0: 32-bit physical address held in $port\_0[31:0]$
Result: the value saved at the physical address
Description: The 32-bit value saved at the given physical address is read out and sent the target FU.

PHY_WRITE
Operand count: 2
Operand 0: 32-bit physical address held in $port\_0[31:0]$
Operand 1: 32-bit value held in $port\_2[31:0]$
Result: void
Description: The 32-bit value provided by operand 1 is written to the given physical address.

SETUP_MULTI_ARRAY
Operand count: 2
Operand 0: 16-bit CTI of a multi-dimensional array type held in $port\_0[15:0]$
Operand 1: 8-bit number of the dimensions of the array type held in $port\_1[7:0]$
Result: void
Description: The operands are loaded into internal registers in order to perform a following `ALLOC_MULTI_ARRAY` operation.

SET_MULTI_ARRAY_DIM_SIZE
Operand count: 1
Operand 0: 32-bit dimension size held in $port\_1[31:0]$
Result: void
Description: The size of a dimension of a multi-dimensional array is loaded into a internal register in order to perform a following `ALLOC_MULTI_ARRAY` operation.